

# HLA Standard Library Reference Manual

1	Passing Parameters to Standard Library Routines . . . . .	1
1.1	Parameter Passing . . . . .	1
1.2	Passing Parameters by Reference and by Value . . . . .	2
1.3	Passing Byte Parameters on the Stack . . . . .	3
1.4	Passing Word Parameters on the Stack . . . . .	5
1.5	Passing DWord Parameters on the Stack . . . . .	5
1.6	Passing QWord Parameters on the Stack . . . . .	6
1.7	Passing TByte Parameters on the Stack . . . . .	7
1.8	Passing LWord Parameters on the Stack . . . . .	7
2	Command-Line Arguments (args.hhf) . . . . .	9
2.1	The Args Module . . . . .	9
2.2	Retrieving the Command Line . . . . .	9
2.3	Argument Count and Item . . . . .	10
2.4	Deleting Command Line Arguments . . . . .	11
2.5	Argument Iterators . . . . .	12
3	Arrays Module (arrays.hhf) . . . . .	15
3.1	The Arrays Module . . . . .	15
3.2	Array Data Types . . . . .	15
3.3	Array Allocation and Deallocation . . . . .	16
3.4	Array Predicates . . . . .	17
3.5	Array Element Access . . . . .	17
3.6	Array Operations . . . . .	18
3.7	Lookup Tables . . . . .	22
4	Bit Manipulation (bits.hhf) . . . . .	25
4.1	Bit Module . . . . .	25
4.2	Bit Counting Function . . . . .	25
4.3	Bit Reversal Functions . . . . .	26
4.4	Bit Merging Functions . . . . .	28
4.5	Bit Extraction Functions . . . . .	30
4.6	Bit Distribution Functions . . . . .	32
5	The Blobs Module (blobs.hhf) . . . . .	35
5.1	Conversion Format Control . . . . .	35

5.2	Blob Synopsis	35
5.3	Blob Internal Representation	36
5.4	Declaring Blob Variables	36
5.4.1	Initializing and Allocating Blob Variables	37
5.5	Blob Accessor Functions	39
5.6	Blob Assignment Functions	42
5.7	Blob Extraction Functions	44
5.8	Blob Comparison Functions	45
5.9	Blob Scanning Functions	45
5.10	Blob Concatenation Functions	48
5.11	Blob Conversion Functions	51
5.12	General Blob I/O Functions	53
5.13	Blob Binary I/O Routines	55
5.14	Blob Output Routines	63
5.15	Blob Input Routines	64
6	Character Classification and Utilities Module (chars.hhf)	65
6.1	Conversion Functions	65
6.2	Predicates (Tests)	66
7	Console Display Control (console.hhf)	73
7.1	The Console Module Module	73
7.2	Cursor Positioning Functions	73
7.3	Console Clearing Functions	76
7.4	Character Insertion/Removal Functions	78
7.5	Console Scrolling	80
7.6	Console Output Colors	81
8	Conversions (conv.hhf)	83
8.1	Buffer vs. String Conversions	83
8.2	Conversion Format Control	84
8.2.1	Underscore Control	84
8.2.2	Delimiter Control	87
8.3	Hexadecimal Conversions	91
8.3.1	Internal Routines	92
8.3.2	Hexadecimal Numeric Size Functions	92
8.3.2.1	Fixed Size Hexadecimal Size Functions	92
8.3.2.2	Standard Hexadecimal Size Functions	96
8.3.3	Hexadecimal Numeric to Buffer Conversions	101
8.3.3.1	Fixed Length Hexadecimal Numeric to Buffer Conversions	101
8.3.3.2	Variable Length Hexadecimal Numeric to Buffer Conversions	108
8.3.4	Hexadecimal Numeric to String Conversions	115
8.3.4.1	Fixed-Length Numeric to Hexadecimal String Conversions	115
8.3.4.2	Variable-Length Numeric to Hexadecimal String Conversions	130
8.3.5	Hexadecimal Buffer to Numeric Conversions	146
8.3.6	Hexadecimal String to Numeric Conversions	151
8.4	Signed Integer Conversions	157
8.4.1	Internal Functions	157
8.4.2	Integer Size Calculations	157

8.4.3	Signed Integer Numeric to Buffer Conversions .....	161
8.4.4	Integer Numeric to String Conversions .....	167
8.4.5	Signed Integer String to Numeric Conversions .....	182
8.5	Unsigned Integer Conversions .....	192
8.5.1	Internal Routines .....	193
8.5.2	Unsigned Integer Size Calculations .....	193
8.5.3	Unsigned Integer Numeric to Buffer Conversions .....	197
8.5.4	Unsigned Integer Numeric to String Conversions .....	203
8.5.5	Unsigned Integer String to Numeric Conversions .....	219
8.6	Floating Point Conversions .....	229
8.6.1	Exponential Floating-Point Conversions .....	229
8.6.2	Floating Point Numeric to Buffer Conversions, Exponential Form ....	230
8.6.3	Floating Point Numeric to String Conversions, Exponential Form ....	232
8.6.4	Floating Point Numeric to Character Conversions, Decimal Form ....	236
8.6.5	Floating-Point Numeric to String Conversions, Decimal Form .....	240
8.6.6	Floating Point String/Buffer to Numeric Conversions .....	244
8.6.7	Roman Numeral Conversion .....	245
9	Coroutines Module (coroutines.hhf) .....	247
9.1	The Coroutine Module .....	247
9.2	The Coroutine Class Definition .....	247
9.3	Coroutine Functions .....	248
10	Character Sets (cset.hhf) .....	251
10.1	Predicates (tests) .....	251
10.2	Character Set Construction and Manipulation .....	259
10.3	Set Operations .....	267
11	Date Functions (datetime.hhf) .....	279
11.1	The Date Module .....	279
11.2	Date Data Types .....	279
11.3	Date Tables .....	280
11.4	Date Predicates .....	281
11.5	Date Conversions .....	283
11.6	Date Arithmetic .....	286
11.7	Reading the Current System Date .....	289
11.8	Date Output and String Conversion .....	290
11.9	Date Class Types .....	291
11.9.1	Date Class Methods/Procedures .....	292
11.9.2	Creating New Date Class Types .....	293
11.9.3	Date Class Functions .....	297
12	Environment Variables Module (env.hhf) .....	301
12.1	The Env Module .....	301
12.2	Retrieving Environment Strings .....	301
13	Exceptions Module (excepts.hhf) .....	303
13.1	The Exceptions Module .....	303
13.2	Exception Resource Reduction .....	303
13.3	Exception Constants .....	303

13.4	Exception Messages . . . . .	310
14	File Class Module (fileclass.hhf) . . . . .	313
14.1	File Class Methods/Procedures . . . . .	313
14.2	A Quick Note . . . . .	314
14.3	General File Operations . . . . .	314
14.4	Opening and Closing Files . . . . .	315
14.5	File Predicates . . . . .	316
14.6	Miscellaneous Output . . . . .	317
14.7	Character, Character Set, and String Output . . . . .	318
14.8	Hexadecimal Numeric Output . . . . .	319
14.9	Signed Integer Numeric Output . . . . .	323
14.10	Unsigned Integer Numeric Output . . . . .	326
14.11	Floating-Point Numeric Output Using Scientific Notation . . . . .	329
14.12	Floating-Point Numeric Output Using Decimal Notation . . . . .	330
14.13	Generic File Output . . . . .	332
14.14	Generic File Input . . . . .	333
14.15	Character and String Input . . . . .	333
14.16	Signed Integer Input . . . . .	334
14.17	Unsigned Integer Input . . . . .	336
14.18	Hexadecimal Input . . . . .	337
14.19	Floating-Point Input . . . . .	339
14.20	Generic File Input . . . . .	339
15	The File I/O Module (fileio.hhf) . . . . .	341
15.1	Conversion Format Control . . . . .	341
15.2	General File I/O Functions . . . . .	341
15.3	File Output Routines . . . . .	354
15.3.1	Miscellaneous Output Routines . . . . .	354
15.3.2	Character, String, and Character Set Output Routines . . . . .	358
15.3.3	Hexadecimal Output Routines . . . . .	365
15.3.4	Signed Integer Output Routines . . . . .	388
15.3.5	Unsigned Integer Output Routines . . . . .	402
15.3.6	Floating Point Output Routines . . . . .	416
15.3.6.1	Real Output Using Scientific Notation . . . . .	416
15.3.6.2	Real Output Using Decimal Notation . . . . .	420
15.3.7	Generic File Output Routine . . . . .	424
15.4	File Input Routines . . . . .	425
15.4.1	General File Input Routines . . . . .	425
15.4.2	Character and String Input Routines . . . . .	428
15.4.3	Signed Integer Input Routines . . . . .	430
15.4.4	Unsigned Integer Input Routines . . . . .	434
15.4.5	Hexadecimal Input Routines . . . . .	437
15.4.6	Floating Point Input . . . . .	442
15.4.7	Generic File Input . . . . .	443
16	The File System Module (filesys.hhf) . . . . .	445
16.1	Filename and Pathname String Functions . . . . .	445
16.2	Directory and File Predicates . . . . .	461

16.3	File Information Functions .....	462
16.4	Directory and File Manipulation Functions.....	463
17	HLA Related Macros and Constants (hla.hhf) .....	469
17.1	The HLA Module.....	469
17.2	Classification Macros .....	469
17.3	String to Integer Macros .....	470
17.4	Label Generation Macro .....	470
17.5	Procedure Overloading Macro.....	470
17.6	Generic PUT Macro.....	473
17.7	@class Constants .....	478
17.8	HLA pType Constants .....	479
17.9	@pclass Return Values .....	481
17.10	@section Return Values .....	482
18	HOWL: The HLA Object Windows Library .....	485
18.1	The HOWL Application Framework.....	485
18.2	The HOWL Declarative Language .....	495
18.2.1	wForm.....	496
18.2.2	wMainMenu..endwMainMenu .....	497
18.2.2.1	wMenuItem .....	498
18.2.2.2	wMenuSeparator .....	498
18.2.2.3	wSubMenu..endwSubMenu.....	498
18.2.2.4	Menu Example.....	498
18.2.3	wTab .....	499
18.2.4	Check Boxes .....	500
18.2.4.1	wCheckable .....	501
18.2.4.2	wCheckBox .....	501
18.2.4.3	wCheckBox3 .....	502
18.2.4.4	wCheckBox3LT.....	502
18.2.4.5	wCheckBoxLT.....	502
18.2.5	wComboBox .....	502
18.2.6	wDragListBox.....	503
18.2.7	wEditBox .....	504
18.2.8	wEllipse .....	505
18.2.9	wIcon .....	506
18.2.10	wGroupBox..endwGroupBox.....	506
18.2.11	wLabel .....	507
18.2.12	wListBox.....	508
18.2.13	wPasswdBox.....	508
18.2.14	wPie .....	509
18.2.15	wPolygon .....	509
18.2.16	wBitmap .....	510
18.2.17	wProgressBar .....	511
18.2.18	wPushButton.....	511
18.2.19	Radio Button Objects .....	511
18.2.19.1	wRadioButton .....	511
18.2.19.2	wRadioButtonLT.....	512
18.2.19.3	wRadioSet..endwRadioSet.....	512

18.2.19.3.1	wRadioSetButton	513
18.2.19.3.2	wRadioSetButtonLT	513
18.2.20	wRectangle	514
18.2.21	wRoundRect	514
18.2.22	wScrollBar	515
18.2.23	wTextEdit	516
18.2.24	wTrackBar	516
18.2.25	wUpDown	517
18.2.26	wUpDownEditBox	518
18.2.27	wTimer	519
18.2.28	wWindow..endwWindow	519
18.3	The HOWL Run-time Library	521
18.3.1	Private Data Fields	523
18.3.2	Abstract Classes	523
18.3.2.1	wBase_t	523
18.3.2.2	wVisual_t	526
18.3.2.3	wClickable_t	529
18.3.2.4	wButton_t	530
18.3.2.5	wCheckable_t	532
18.3.2.6	wSurface_t	533
18.3.2.7	wFilledFrame_t	534
18.3.2.8	wabsEditBox_t	535
18.3.2.9	wContainer_t	539
18.3.3	Containers	541
18.3.3.1	Forms and Menus	541
18.3.3.1.1	wForm_t	541
18.3.3.1.2	wMenu_t	543
18.3.3.1.3	wMenuItem_t	543
18.3.3.2	Tabbed Forms	545
18.3.3.2.1	wTabs_t	545
18.3.3.3	wGroupBox_t	547
18.3.4	Graphic Objects	548
18.3.4.1	wBitmap_t	548
18.3.4.2	wEllipse_t	550
18.3.4.3	wPie_t	551
18.3.4.4	wPolygon_t	553
18.3.4.5	wRectangle_t	555
18.3.4.6	wRoundRect_t	556
18.3.5	Buttons	557
18.3.6	wCheckBox_t	557
18.3.7	wCheckBox3_t	558
18.3.8	wCheckBox3LT_t	559
18.3.9	wCheckBoxLT_t	560
18.3.10	wPushButton_t	561
18.3.11	wRadioButton_t	561
18.3.12	wRadioButtonLT_t	562
18.3.13	wRadioSet_t	563
18.3.13.1	wRadioSetButton_t	564

18.3.13.2	wRadioSetButtonLT_t	565
18.3.14	Editors and Edit Boxes	566
18.3.14.1	wEditBox_t	566
18.3.14.2	wPasswdBox_t	567
18.3.14.3	wTextEdit_t	568
18.3.15	List, Drag, and Combo Boxes	570
18.3.15.1	wListBox_t	570
18.3.15.2	wDragListBox_t	573
18.3.15.3	wComboBox_t	574
18.3.16	Progress Bars	576
18.3.16.1	wProgressBar_t	576
18.3.17	Scroll Bars and Track Bars	577
18.3.17.1	wScrollBar_t	578
18.3.17.2	wTrackBar_t	583
18.3.18	Up/Down Arrows	587
18.3.18.1	wUpDown_t	587
18.3.18.2	wUpDownEditBox_t	589
18.3.19	Icons	591
18.3.19.1	wIcon_t	591
18.3.20	Text	592
18.3.20.1	wFont_t	592
18.3.20.2	wLabel_t	594
18.3.21	Views, Windows, and Tab Pages	597
18.3.21.1	wTabPage_t	597
18.3.21.2	wView_t	599
18.3.21.3	window_t	599
18.3.22	Timers	601
18.3.22.1	wTimer_t	601
19	The Linux Module (linux.hhf)	605
19.1	The Linux Module	605
19.2	The Linux Header File	605
20	Lists Module (lists.hhf)	607
20.1	The Lists Module	607
20.1.0.1	List Data Types	607
20.2	List_t Class Function Types	609
20.3	Creating New List Class Types	610
20.4	List Procedures, Methods, and Iterators	611
20.5	List Constructor and Destructor	612
20.6	Accessor Functions	613
20.6.0.1	Adding Nodes to a List	613
20.7	Removing Nodes From a List	615
20.8	Accessing Nodes in a List	617
20.9	Miscellaneous List Functions	619
21	Math Module (math.hhf)	623
21.1	The Math Module	623
21.2	Math Data Types	623

21.3	64-Bit Arithmetic and Logical Operations .....	623
21.4	128-Bit Arithmetic and Logical Operations .....	631
21.5	Transcendental, Logarithmic, and Other Floating-Point Operations.....	640
22	Memory-Mapped I/O (mmap.hhf) .....	667
22.1	MMAP Module .....	667
22.2	Class Fields .....	667
22.3	Class Procedures and Methods .....	667
23	Memory (memory.hhf) .....	671
23.1	Memory Module .....	671
23.2	Deprecated Names .....	671
23.3	Generic Memory Allocation .....	671
23.4	String Memory Allocation.....	678
24	OS Module (os.hhf) .....	681
24.1	The OS Module .....	681
24.2	Executing Shell Commands.....	681
24.3	Delaying Program Execution.....	682
25	Patterns Module (patterns.hhf) .....	683
25.1	The Patterns Module .....	683
25.2	An Introduction to Pattern Matching (a tutorial).....	683
25.3	Pattern Matching Functions Versus User Code.....	687
25.4	Register and Stack Usage in Pattern Matching Statements.....	688
25.5	Nesting Pattern Matching Statements .....	690
25.6	Cleanly Nesting Patterns .....	694
25.7	Backtracking .....	696
25.8	Pattern Components.....	698
25.9	Lazy / Eager Evaluation and Pattern Matching Performance .....	699
25.10	Regular Expressions .....	700
25.11	Pattern Matching Statements .....	705
25.12	Alternation .....	706
25.13	Pattern Matching Macros.....	707
25.14	Character Set Matching Functions.....	710
25.15	Character Matching Functions.....	715
25.16	Case Insensitive Character Matching Routines .....	720
25.17	String Extraction Functions .....	726
25.18	Whitespace and End of String Matching Functions .....	727
25.19	Matching an Arbitrary Sequence of Characters.....	728
25.20	Writing Your Own Pattern Matching Routines .....	729
26	Random Number Generator Module (rand.hhf) .....	741
26.1	The Random Module .....	741
26.2	The Random Number Generators .....	741
27	RPC: The Remote Procedure Calls Library .....	745
27.1	Types of Remote Procedures .....	745
27.2	Remote Procedure Call (RPC) Protocol .....	745
27.3	The RPC Declaration Language (RDL).....	746



27.4	RPC Preliminaries . . . . .	749
27.5	RPC Clients . . . . .	749
27.6	RPC Servers . . . . .	751
27.7	Passing Large Objects Between the Client and Server . . . . .	755
28	Sockets Module (sockets.hhf) . . . . .	761
28.1	The SOCK Module . . . . .	761
28.2	Socket Initialization and Cleanup . . . . .	761
28.3	Generic Socket Functions . . . . .	761
28.4	Low-Level BSD-Style Socket Functions . . . . .	762
28.5	Socket Classes . . . . .	770
28.6	A Quick Note . . . . .	771
28.7	Client/Server Applications Using the Socket Classes . . . . .	771
28.8	A Simple Server Application . . . . .	771
28.9	A Simple Client Application . . . . .	774
28.10	Client/Server Communication . . . . .	774
28.11	General Socket Class Operations . . . . .	775
28.12	Miscellaneous Output . . . . .	778
28.13	Character, Character Set, and String Output . . . . .	779
28.14	Hexadecimal Numeric Output . . . . .	781
28.15	Signed Integer Numeric Output . . . . .	785
28.16	Unsigned Integer Numeric Output . . . . .	788
28.17	Floating-Point Numeric Output Using Scientific Notation . . . . .	790
28.18	Floating-Point Numeric Output Using Decimal Notation . . . . .	792
28.19	Generic File Output . . . . .	793
28.20	Generic File Input . . . . .	794
28.21	Character and String Input . . . . .	795
28.22	Signed Integer Input . . . . .	796
28.23	Unsigned Integer Input . . . . .	797
28.24	Hexadecimal Input . . . . .	799
28.25	Floating-Point Input . . . . .	800
28.26	Generic File Input . . . . .	801
29	The Standard Error Module (stderr.hhf) . . . . .	803
29.1	Conversion Format Control . . . . .	803
29.2	File I/O Routines and the Standard Error Handle . . . . .	803
29.3	Standard Error Routines . . . . .	803
29.4	Miscellaneous Output Routines . . . . .	804
29.5	Boolean Output . . . . .	805
29.6	Character, String, and Character Set Output Routines . . . . .	806
29.7	Hexadecimal Output Routines . . . . .	812
29.8	Signed Integer Output Routines . . . . .	834
29.9	Unsigned Integer Output Routines . . . . .	847
29.10	Floating Point Output Routines . . . . .	861
29.10.1	Real Output Using Scientific Notation . . . . .	861
29.10.2	Real Output Using Decimal Notation . . . . .	864
29.11	Generic Error Output Routine . . . . .	869
30	The Standard Input Module (stdin.hhf) . . . . .	871

30.1	Conversion Format Control . . . . .	871
30.2	File I/O Routines and the Standard Output Handle . . . . .	871
30.3	Standard Input Routines . . . . .	871
30.4	General Standard Input Routines . . . . .	872
30.5	Character and String Input Routines . . . . .	874
30.6	Hexadecimal Input Routines . . . . .	875
30.7	Signed Integer Input Routines . . . . .	878
30.8	Unsigned Integer Input Routines . . . . .	880
30.9	Floating Point Input . . . . .	883
30.10	Generic File Input . . . . .	883
31	The Standard Output Module (stdout.hhf) . . . . .	885
31.1	Conversion Format Control . . . . .	885
31.2	File I/O Routines and the Standard Output Handle . . . . .	885
31.3	Standard Output Routines . . . . .	885
31.4	Miscellaneous Output Routines . . . . .	886
31.5	Boolean Output . . . . .	887
31.6	Character, String, and Character Set Output Routines . . . . .	888
31.7	Hexadecimal Output Routines . . . . .	895
31.8	Signed Integer Output Routines . . . . .	917
31.9	Unsigned Integer Output Routines . . . . .	930
31.10	Floating Point Output Routines . . . . .	943
	31.10.1 Real Output Using Scientific Notation . . . . .	943
	31.10.2 Real Output Using Decimal Notation . . . . .	946
31.11	Generic Standard Output Routine . . . . .	951
32	The HLA Standard Template Library . . . . .	953
32.1	Introduction to the HLA STL . . . . .	953
32.2	Type Declarations Created by a Template . . . . .	954
32.3	Template Objects are Classes . . . . .	954
32.4	Class Traits . . . . .	955
	32.4.1 isSTL_c Trait . . . . .	955
	32.4.2 Compile-Time Traits . . . . .	956
	32.4.3 Run-Time Traits . . . . .	956
	32.4.4 Trait Constants . . . . .	957
	32.4.4.1 stl.isContainer_c Trait . . . . .	957
	32.4.4.2 stl.isRandomAccess_c Trait . . . . .	958
	32.4.4.3 stl.isArray_c Trait . . . . .	958
	32.4.4.4 stl.isVector_c Trait . . . . .	958
	32.4.4.5 stl.isDeque_c Trait . . . . .	958
	32.4.4.6 stl.isList_c Trait . . . . .	958
	32.4.4.7 stl.isTable_c Trait . . . . .	958
	32.4.4.8 stl.supportsOutput_c Trait . . . . .	958
	32.4.4.9 stl.supportsCompare_c Trait . . . . .	958
	32.4.4.10 stl.supportsInsert_c Trait . . . . .	958
	32.4.4.11 stl.supportsRemove_c Trait . . . . .	958
	32.4.4.12 stl.supportsAppend_c Trait . . . . .	958
	32.4.4.13 stl.supportsPrepend_c Trait . . . . .	958
	32.4.4.14 stl.supportsForEach_c and supportsrForEach_c Traits . . . . .	959

32.4.4.15	stl.supportsCursor_c Trait	959
32.4.4.16	stl.supportsSearch_c Trait	959
32.4.4.17	stl.supportsElementSwap_c Trait	959
32.4.4.18	stl.supportsObjSwap_c Trait	959
32.4.4.19	stl.elementsAreObjects_c Trait	959
32.4.4.20	stl.fastInsert_c Trait	959
32.4.4.21	stl.fastRemove_c Trait	959
32.4.4.22	stl.fastAppend_c Trait	959
32.4.4.23	stl.fastPrepend_c Trait	959
32.4.4.24	stl.fastSwap_c Trait	960
32.4.4.25	stl.fastSearch_c Trait	960
32.4.4.26	stl.fastElementSwap_c Trait	960
32.4.5	Other Run-Time Traits	960
32.5	The Vector Template	960
32.6	The Deque Template	960
32.7	The List Template	960
32.8	The Table Template	960
33	The Strings Module (strings.hhf)	961
33.1	The HLA String Data Type	961
33.2	String Allocation Macros and Functions	963
33.3	String Length Calculations	964
33.4	String Assignment	964
33.5	Substring Functions	966
33.6	String Insertion and Deletion Functions	972
33.7	String Comparison Functions	980
33.8	String Searching Functions	986
33.9	Character Set Searching Functions	999
33.10	String Parsing Functions	1005
33.11	String Formatting Functions	1021
33.12	String Conversion Functions	1033
33.13	String Concatentation Functions	1040
33.14	String Value Concatentation Functions	1047
33.14.1	Boolean Output	1047
33.14.2	Character, String, and Character Set Concatenation Routines	1048
33.14.3	Hexadecimal Concatenation Routines	1053
33.14.4	Signed Integer Concatenation Routines	1068
33.14.5	Unsigned Integer Concatenation Routines	1078
33.15	Floating-Point Concatenation Routines	1088
33.15.1	Real to String Output Using Scientific Notation	1089
33.15.2	Real To String Output Using Decimal Notation	1091
33.16	Generic String Format Output Routine	1095
34	High-Level Language Module (hll.hhf)	1097
34.1	The HLL Module	1097
34.2	The switch/case/default/endswitch Macro	1097
35	Tables Module (tables.hhf)	1099
35.1	The Tables Module	1099

35.2	The Table Class .....	1099
36	Threads Module (threads.hhf) .....	1103
36.1	Threads Module .....	1103
36.2	Thread Creation .....	1103
36.3	Thread Identification .....	1104
36.4	Thread Local Storage .....	1105
36.5	Events .....	1107
36.6	Critical Sections .....	1108
36.7	Semaphores .....	1110
37	Time Functions (datetime.hhf) .....	1113
37.1	Time Module .....	1113
37.2	Time Data Types .....	1113
37.3	Time Predicates .....	1114
37.4	Time Conversions .....	1116
37.5	Time Arithmetic .....	1119
37.6	Reading the Current System Time .....	1122
37.7	Time String Conversions and Output .....	1123
37.8	Time Class Types .....	1124
37.9	Time Class Methods/Procedures .....	1124
37.10	Creating New Time Class Types .....	1125
37.11	Time Class Functions .....	1129
38	Timer Class and Module (timer.hhf) .....	1133
38.1	Timer Module .....	1133
38.2	Timer Class/Data Structure .....	1133
38.3	Timer Operation .....	1134
38.4	Timer Class Fields .....	1134
38.5	Timer Procedures and Methods .....	1134
39	Zero-terminated String Functions (zstring.hhf) .....	1137
39.1	ZStrings Module .....	1138
39.2	Zstring Functions .....	1138

# 1 Passing Parameters to Standard Library Routines

## 1.1 Parameter Passing

Standard library functions typically compute some value based on a set of input values. Understanding how to pass these *parameter values* to the standard library routines is important if you want to make efficient calls to the library routines.

Most standard library functions expect their parameters in one of two locations – in one or more registers or on the stack (or both). Generally, standard library functions preserve all the general-purpose register values across a call, with the exception of those functions that explicitly return a value in a register (or multiple registers, if needed). Therefore, the standard library functions are careful about passing values to a function in a register because such semantics might require the caller to use a register to hold a parameter value that it is already using for a different purpose. However, if a function returns some result in a register, then the standard library routines assume that the register's contents are fair game on input and may specify the use of that register as an input parameter. In almost every other case, the standard library routines expect the caller to pass parameters on the stack, though there are a few exceptions to this rule.

In most cases, the standard library routines only use a register to pass a parameter when there is exactly one input parameter and one function return result. Typically, the functions will use the EAX register as both the input and output value. In a few rare cases, a function may have two or more parameters with one parameter being passed in a register and the remaining parameters being passed on the stack.

If you are using a high-level calling syntax, you should take care when calling routines that pass multiple parameters in registers. Consider the `conv.bToBuf` (byte/hex string conversion to buffer) function:

```
procedure conv.bToBuf( b:byte in al; var buffer:var in edi );
```

HLA will automatically emit code that loads the registers when you call this function. E.g., the following

```
conv.bToBuf( b, myBuffer );
```

will emit the following code:

```
mov( b, al );
lea( edi, buffer );
call conv.bToBuf;
```

Suppose, however, that you write code like the following:

```
conv.bToBuf( b, [eax] ); // EAX points at the buffer
```

In this case, HLA will generate the following code, which will probably not do what you want:

```
mov( bl, al ); // Load b parameter into AL, as before
mov( eax, edi ); // EAX's value was munged by the instruction above
call conv.bToBuf;
```

While the problem is obvious when writing low-level code, the high-level invocation hides the problem. This is one drawback to using a high-level invocation of the library code: it tends to hide what's going on and problems like this, though they are very rare, are more easily spotted using low-level code. In defense of the high-level invocation style, it catches far more *common* errors than it misses, so this isn't sufficient reason to avoid using high-level invocations.

The correct solution, by the way, is to always be aware of where the standard library routines pass their parameters. When the standard library passes a given parameter in a register, you should attempt to have that value sitting in the register when you call the function. This is safest and generates the most efficient code. For example, consider the following call to the `conv.bToBuf` library routine:

```
conv.bToBuf( al, [edi] );
```

Because the parameter data is already sitting in the registers where `conv.bToBuf` expects them, this generates the following (very efficient) code:

```
call bToBuf;
```

## 1.2 Passing Parameters by Reference and by Value

The standard library routines generally employ one of two different parameter passing mechanisms – pass by value and pass by reference. Pass by value, as its name implies, passes the value of a parameter to a function. For small objects (say, less than 16 bytes or so), pass by value is very efficient. For large objects (e.g., arrays or records larger than 16 bytes or so) the caller must make a copy of the data when passing it to a subroutine, this data copy operation can be very slow if the data object is large. Therefore, the standard library does not use pass by value when passing arrays, records, or other large data structures to a library routine.

Note that the decision to use pass by reference or pass by value is entirely up to the routine's designer. If you're calling a standard library routine you must use the same parameter passing mechanism the designer used when writing the function. The function's documentation will tell you whether a parameter is passed by reference or by value (or you can read the HLA prototype for a function which will also tell you the parameter passing mechanism).

Pass by reference parameters pass the 32-bit *address* of the actual parameter to their corresponding function. Because an address is always 32 bits, regardless of the actual parameter data size, passing a large parameter by reference can be far more efficient than passing that same parameter by value. Another benefit (or drawback, in certain cases) to using pass by reference parameters is that the function can modify the value of the actual parameter object because it has the memory address of that object. For more details on the semantics of pass by reference versus pass by value, check out the chapter on procedures and functions in *The Art of Assembly Language*.

When passing a parameter by reference to a function, you must first compute the address of that object and pass the address to the function. For example, consider the following function prototype:

```
procedure someFunction( var refParm:byte );
```

(for those unfamiliar with HLA syntax, the "var" prefix tells HLA that refParm is a pass by reference parameter.) To call someFunction, you must push the address of the actual parameter onto the stack. If the parameter is a static object (e.g., an HLA STATIC, READONLY, or STORAGE variable), then you can compute the address using the HLA "&" (address-of operator), thusly:

```
pushd( &actualByteParameter );
call someFunction;
```

If the actual parameter is an automatic variable, or you reference it using some complex addressing mode (other than displacement-only), then you'd use code like the following to pass actualByteParameter to someFunction:

```
lea( eax, actualByteParameter );
push( eax );
call someFunction;
```

Note that this scheme, unfortunately, makes use of a general-purpose 32-bit register.

Of course, you can use the HLA high-level function invocation syntax to automatically generate the calling sequence for you:

```
someFunction( actualByteParameter );
```

HLA is smart enough to determine the storage class of the actualByteParameter object and generate appropriate code to pass that parameter's address on the stack. Note that, unlike the example given earlier, HLA does not wipe out the value in the EAX register if it needs to compute the parameter's address with an LEA instruction; HLA will always preserve all register values unless you explicitly state that you're passing the parameter in a 32-bit register. For simple local variables or other variables allocated on the stack (e.g., parameters passed into the current function), HLA will actually generate code like the following:

```
push( ebp ); // Assumes EBP points at the current stack frame
add( @offset( actualByteParameter ), (type dword [esp]));
```

This pushes the address of a local variable or parameter onto the stack without affecting any general-purpose register values (other than ESP, which we expect).

Note that registers do not have addresses. Therefore, you cannot pass a register by reference to a function; i.e., the following is always illegal:

```
someFunction( edi );
```

Of course, what most programmer's mean when they attempt something like this is that EDI contains the address of the variable to pass to the function. However, HLA assumes that you're trying to take the address of EDI, which is always illegal. The quick and dirty way to handle this issue is to explicitly tell HLA that EDI is a pointer to the data using an invocation like this:

```
someFunction( [edi] );
```

This approach works great when the 32-bit address appears in a register. In the more general case, the 32-bit value could appear in any 32-bit object (e.g., in a pointer variable). You can use HLA's VAL prefix to explicitly tell HLA to treat a 32-bit value as an address to be passed as a reference parameter, e.g.,

```
someFunction( val edi );
someFunction( val dwordVar );
```

HLA is actually smart enough to recognize certain special cases where you're trying to pass the contents of a pointer variable as a reference parameter. Consider the following HLA code fragment:

```
static
  ptrVar :pointer to byte := &someStaticByteVar;
  .
  .
  .
  someFunction( ptrVar );
```

HLA realizes that ptrVar is a pointer to a byte object and will automatically push ptrVar's value onto the stack rather than generating an error. No explicit VAL will be necessary. Note that the base type of the pointer variable must match the reference parameter's type for this call to be successful.

HLA supports a special "untyped pass by reference parameter" syntax. The following example demonstrates this:

```
procedure untypedRefParm( var anyMemoryType: var );
```

Note that the parameter does not have an explicit type. The keyword "var" tells the compiler that the parameter is an untyped reference parameter and any memory variable can be passed to this function.

If you pass a variable name as an actual parameter for an untyped reference parameter, HLA will always take the address of that object, regardless of its type. If you want to pass the value of a pointer variable, rather than the address of that pointer variable, then you must explicitly supply the VAL prefix, e.g.,

```
untypedRefParm( val ptrVal );
```

HLA string objects are hybrid pointer/value objects. An HLA string variable is a dword object that contains a pointer to the actual string data. Consider, though, the following HLA procedure declaration:

```
procedure someStrFunction( s:string );
```

This function has a single pass-by-value parameter. It might seem confusing that this is a pass-by-value object as it passes the address of the actual character data that makes up the string to the function. However, keep in mind that a string object is a pointer and what you're really passing by value is the value of that pointer variable. If you were to pass a string object by reference, what you would be passing is the address of the string variable (that is, the address of the address of the actual character data). For this reason, if a 32-bit register contains the value of a string variable (that is, the register contains the address of the actual character data), then the following call to someStrFunction is perfectly legitimate:

```
someStrFunction( eax );
```

## 1.3 Passing Byte Parameters on the Stack

For efficiency reasons, standard library routines always pass all parameters as a multiple of four bytes. When passing a byte-sized parameter on the stack by value, the actual parameter value consumes the L.O. byte of the double word passed on the stack. The function ignores the H.O. three bytes of the value passed for this parameter, though by convention (to make debugging a little easier) you should try to pass zeros in the H.O. three bytes if it is not inconvenient to do so.

When passing a byte-sized constant, you should simply push the dword containing the 8-bit value, e.g.,

```
pushd( 5 );
call someLibraryRoutine;
```

When passing the 8-bit value of the 8-bit registers AL, BL, CL or DL onto the stack, you should simply push the 32-bit register that holds the 8-bit register, e.g.,

```
push( eax ); // Pushes AL onto the stack
call someLibraryRoutine;
push( ebx ); // Pushes BL onto the stack
call someOtherLibraryRoutine;
```

Note that this trick does not apply to the AH, BH, CH, or DH registers. The best code to use when you need to push these registers is to drop the stack down by four bytes and then move the desired register into the memory location you've just created on the stack, e.g.,

```
sub( 4, esp );
mov( AH, [esp] ); // Pushes AH onto the stack
call someLibraryRoutine;
sub( 4, esp );
mov( BH, [esp] ); // Pushes BH onto the stack
call anotherLibraryRoutine;
```

Here's another way you can accomplish this (a little slower, but leaves zeros in the H.O. three bytes):

```
pushd( 0 );
mov( CH, [esp] ); // Pushes CH onto the stack
call someLibraryRoutine;
```

When passing a byte-sized variable, you should try to push the variable's value and the following three bytes, using code like the following (HLA syntax):

```
pushd( (type dword eightBitVar) );
call someLibraryRoutine;
```

There is one drawback to the approach above. In three very rare cases the code above could cause a segmentation fault. If the 8-bit variable is located on the last three bytes of a page in memory (4,096 bytes) and the next memory page is not readable, the system will generate a fault if you attempt to push all four bytes. In such a case, the next best solution, if a register is available, is to move the 8-bit value into AL, BL, CL, or DL and push the corresponding 32-bit register. If no registers are available, then you can write code like the following:

```
push( eax );
push( eax );
mov( byteVar, al );
mov( al, [esp+4] );
pop( eax );
call someLibraryRoutine;
```

This code is ugly and slightly inefficient, but it will always work (assuming, of course, you don't get a stack overflow).

The HLA compiler will generate code similar to this last example if you pass a byte variable as the actual parameter to a library function expecting an 8-bit value parameter:

```
someLibraryRoutine( byteVar );
```

Therefore, if efficiency is a concern to you, you should try to load the byte variable (byteVar in this example) into AL, BL, CL, or DL prior to calling someLibraryRoutine, e.g.,

```
mov( boolVar, al );
someLibraryRoutine( al );
```

Another solution, if you want to use an HLA high-level-like calling sequence, is to use a hybrid calling sequence and explicitly specify the instruction(s) to use to pass a byte-sized parameter on the stack. For example,

```
someLibraryRoutine( #{ push( (type dword boolVar) ); }# );
```



Unfortunately, you lose the benefit of type checking and other semantic checks the compiler would normally do for you when using this hybrid syntax. Nevertheless, this scheme does have the advantage of encapsulating the parameter pushing code into a single sequence, so that it's obvious which instruction(s) go with the particular parameter. This is not the case when you manually push the parameters onto the stack as in the earlier examples.

## 1.4 Passing Word Parameters on the Stack

For efficiency reasons, standard library routines always pass all parameters as a multiple of four bytes. When passing a word-sized parameter on the stack by value, the actual parameter value consumes the L.O. two bytes of the double word passed on the stack. The function ignores the H.O. word of the value passed for this parameter, though by convention (to make debugging a little easier) you should try to pass zeros in the H.O. word if it is not inconvenient to do so.

When passing a word-sized constant, you should simply push the double word containing the 16-bit value, e.g.,

```
pushd( 5 );
call someLibraryRoutine;
```

When passing the 16-bit value of a 16-bit register (AX, BX, CX, DX, SI, DI, BP, or SP) onto the stack, you should simply push the 32-bit register that holds the 16-bit register, e.g.,

```
push( eax ); // Pushes AX onto the stack
call someLibraryRoutine;
push( ebx ); // Pushes BX onto the stack
call someOtherLibraryRoutine;
```

When passing a word-sized variable, you should try to push the variable's value and the following two bytes, using code like the following (HLA syntax):

```
pushd( (type dword sixteenBitVar) );
call someLibraryroutine;
```

There is one drawback to the approach above. In three very rare cases the code above could cause a segmentation fault. If the 16-bit variable is located on the last three bytes of a page in memory (4,096 bytes) and the next memory page is not readable, the system will generate a fault if you attempt to push all four bytes. In such a case, the next best solution, is to use two consecutive pushes:

```
pushw( 0 ); // H.O. word is zeros
push( sixteenBitVar );
call someLibraryRoutine;
```

The HLA compiler will generate code similar to this last example if you pass a word variable as the actual parameter to a library function expecting a 16-bit value parameter:

```
someLibraryRoutine( wordVar );
```

## 1.5 Passing DWord Parameters on the Stack

Because 32-bit dword objects are the native x86 data type, there are only a few issues with passing 32-bit parameters on the stack to a standard library routine.

First of all, and this applies to all stack operations not just 32-bit pushes and pops, you should always keep the stack 32-bit aligned. That is, the value in ESP should always contain a value that is a multiple of four (i.e., the L.O. two bits of ESP must always contain zeros). If this is not the case, many standard library routines will fail.

When passing a 32-bit value onto the stack, just about any mechanism you can use to push that value is perfectly valid. You can efficiently push constants, registers, and memory locations using a single push instruction, e.g.,

```
pushd( 12345 ); // Passing a 32-bit constant
push( mem32 ); // Passing a dword variable
push( eax ); // Passing a 32-bit register
call someLibraryRoutine;
```

One type of double word parameter deserves special mention – a reference parameter. Reference parameters pass the address of their object on the stack rather than the value of the object (that is, they pass a pointer to the actual object). HLA actually supports several different reference parameter types including pass by reference (VAR) parameters, pass by result (RESULT) parameters, and pass by value/returned (VALRES) parameters. Of these three parameter types, the standard library only uses the VAR (pass by reference) type, though if you ever see a function that uses one of these other parameter mechanisms the calling sequence is exactly the same.

When passing a parameter by reference, you must push the address of the actual parameter (rather than its value) onto the stack. For static objects, you can use the push immediate instruction, e.g., (in HLA syntax):

```
pushd( &staticVar );
call someLibraryRoutine;
```

For automatic variables, or objects whose address is not a simple static offset (e.g., a complex pointer address involving registers and what-not), you'll have to use the LEA instruction to first compute the address and then push that register's value, e.g.,

```
lea( eax, anAutomaticVar ); // Variable allocated on the stack
push( eax );
call someLibraryRoutine;
```

If the variable's address is a simple offset from a single register (such as automatic variables declared in the stack frame and referenced off of the EBP register), you can push the address of the variable by pushing the base register and adding the offset of that variable to the value left on the stack, thusly:

```
push( ebp ); // anAutoVar is found at EPB+@offset(anAutoVar)
add( @offset( anAutoVar ), (type dword [esp]));
call someLibraryRoutine;
```

If the address you want to pass in a reference parameter is a complex address, you'll have to use the LEA instruction to compute that address and push it onto the stack. This, unfortunately, requires a free 32-bit register. If no 32-bit registers are free, you can use code like the following to achieve this:

```
sub( 4, esp ); // Reserve space for parameter on stack
push( eax ); // Preserve EAX
lea( eax, [ebp+@offset(autoVar)][ecx*4+3] );
mov( eax, [esp+4] ); // Store in parameter location
pop( eax ); // Restore EAX
call someLibraryRoutine;
```

Of course, it's much nicer to use the HLA high-level syntax for calls like this as the HLA compiler will automatically handle all the messy code generation details for you.

## 1.6 Passing QWord Parameters on the Stack

Because qword (64-bit) objects are a multiple of 32 bits in size, manually passing qword objects on the stack is very easy. All you need do is push two dword values. Because the stack grows downward in memory and the x86 is a little endian machine, you must push the H.O. dword first and the L.O. dword second.

If the qword value is held in a register pair, then push the register containing the H.O. dword first and the L.O. dword second. For example, if EDX:EAX contains the 64-bit value, then you'd push the qword as follows:

```
push( edx ); // Push H.O. dword
push( eax ); // Push L.O. dword
call someLibraryRoutine;
```

If the qword value is held in a qword variable, then you must first push the H.O. dword of that variable followed by the L.O. dword, e.g.,

```
push( (type dword qwordVar[4])); // Push H.O. dword first
push( (type dword qwordVar)); // Push L.O. dword second
call someLibraryRoutine;
```

If the qword value you wish to pass is a constant, then you've got to compute the L.O. and H.O. dword values for that constant and push those. When using HLA, you can use the compile-time computational capabilities of HLA to do this for you, e.g.,

```

pushd( ((some64bitConst) >> 32);
pushd( ((some64bitConst) & $FFFF_FFFF );
call someLibraryRoutine;

```

If this is something you do frequently, you might want to create a macro to break up the 64-bit value and push it for you.

Of course, you can always use the HLA high-level syntax to pass a 64-bit object to a standard library routine. HLA automatically generates the appropriate code to pass the qword object as a parameter on the stack.

## 1.7 Passing TByte Parameters on the Stack

For efficiency reasons, standard library routines always pass all parameters as a multiple of four bytes. When passing a byte-sized parameter on the stack by value, the actual parameter value consumes the L.O. ten bytes of the three double words passed on the stack. The function ignores the H.O. word of the value passed for this parameter, though by convention (to make debugging a little easier) you should try to pass zeros in the H.O. word if it is not inconvenient to do so.

The following code demonstrates how to pass a ten-byte object to a standard library routine:

```

pushw( 0 ); // Dummy H.O. word of zero
push( (type word tbyteVar[8])); // Push H.O. byte of tbyte object
push( (type dword tbyteVar[4])); // Push bytes 4-7 of tbyte object
push( (type dword tbyteVar[0])); // Push L.O. dword of tbyte object
call someLibraryRoutine;

```

If your tbyte object is not at the very end of allocated memory, you could probably combine the first two instructions in this sequence to produce the following (slightly more efficient) code:

```

push( (type dword tbyteVar[8])); // Pushes two extra bytes.

```

This pushes the two bytes beyond tbyteVar onto the stack, but presumably the function will ignore all bytes beyond the tenth byte passed on the stack, so the actual values in those H.O. two bytes are irrelevant. Note the earlier discussion (in the section on pushing byte parameters) about the rare possibility of a memory access error when using this trick.

Of course, you can always use the HLA high-level syntax to pass an 80-bit object to a standard library routine. HLA automatically generates the appropriate code to pass the tbyte object as a parameter on the stack.

## 1.8 Passing LWord Parameters on the Stack

Because lword (128-bit) objects are a multiple of 32 bits in size, manually passing lword objects on the stack is very easy. All you need do is push four dword values. Because the stack grows downward in memory and the x86 is a little endian machine, you must push the H.O. dword first and the L.O. dword last.

If the lword value is held in an lword variable, then you must first push the H.O. dword of that variable followed by the lower-order dwords, down to the L.O. dword, e.g.,

```

push( (type dword qwordVar[12])); // Push H.O. dword first
push( (type dword qwordVar[8])); // Push bytes 8-11 second
push( (type dword qwordVar[4])); // Push bytes 4-7 third
push( (type dword qwordVar)); // Push L.O. dword last
call someLibraryRoutine;

```

If the lword value you wish to pass is a constant, then you've got to compute the four dword values for that constant and push those. When using HLA, you can use the compile-time computational capabilities of HLA to do this for you, e.g.,

```

pushd( ((some128bitConst) >> 96);
pushd( ((some128bitConst) >> 64 & $FFFF_FFFF );
pushd( ((some128bitConst) >> 32 & $FFFF_FFFF );
pushd( ((some128bitConst) & $FFFF_FFFF );
call someLibraryRoutine;

```

If this is something you do frequently, you might want to create a macro to break up the 128-bit value and push it for you.

Of course, you can always use the HLA high-level syntax to pass a 128-bit object to a standard library routine. HLA automatically generates the appropriate code to pass the lword object as a parameter on the stack.

## 2 Command-Line Arguments (args.hhf)

The HLA args module provides access to, and support for, Windows Command Line Interpreter or Linux/ \*nix shell command line parameters.

Note: be sure to read the chapter on "Passing Parameters to Standard Library Routines" (parmpassing.rtf) before reading this chapter.

**A Note About Thread Safety:** The args module maintains a couple of static global variables that maintain the command-line values. Currently, these values apply to all threads in a process. You should take care when changing these values in threads. The command-line is a resource that must be shared amongst all threads in an application. If you write multi-threaded applications, it is your responsibility to serialize access to the command-line functions.

### 2.1 The Args Module

To use the args functions in your application, you will need to include one of the following statements at the beginning of your HLA application:

```
#include( "args.hhf" )
or
#include( "stdlib.hhf" )
```

### 2.2 Retrieving the Command Line

The `arg.cmdLn` and `arg.a_cmdLn` functions retrieve the entire command-line as a string. This string typically consists of the list of command-line parameters, each parameter separated by a single space. Note that on some operating systems, the HLA standard library command-line functions might actually synthesize this string by concatenating the command-line arguments together, so you can not expect this string to be an exact representation of the command line that the user typed (that is, the user may have typed extra spaces or other delimiter characters that the OS' shell discarded before passing the command line text to the program). Also, the command-line string will not contain I/O redirection or other process-related items (e.g., pipes) found on the command line.

**procedure arg.cmdLn();**

*arg.cmdLn* returns a pointer to a static string held inside the stdlib code. This function returns the string pointer in EAX. The caller must not modify any data in this string (use `arg.a_cmdLn` if you need a malleable string).

HLA high-level calling sequence examples:

```
arg.cmdLn();
stdout.put( "Command line: ", (type string eax), nl );
```

HLA low-level calling sequence examples:

```
call arg.cmdLn;
push( eax );
call stdout.puts;
```

**procedure arg.a\_cmdLn();**

*arg.a\_cmdLn* returns, in EAX, a string containing a copy of the command-line parameter text. This is a pointer to a string allocated on the heap. It is the caller's responsibility to free this storage (via a call to `str.free`) when it is done using the string.

HLA high-level calling sequence examples:

```
arg.a_cmdLn();
stdout.put( "Command line: ", (type string eax), nl );
str.free( eax );
```

HLA low-level calling sequence examples:

```
call arg.cmdLn;
push( eax );
call stdout.puts;
push( eax );
call str.free;
```

## 2.3 Argument Count and Item

The functions in this category are the standard argument functions that most programs use: `arg.c`, `arg.v`, and `arg.a_v`. The `arg.c` function returns the number of command-line parameters and `arg.v`/`arg.a_v` return a pointer to a string that corresponds to an individual parameter.

### **procedure arg.c();**

*arg.c* returns the number of command-line parameters in EAX. This count includes the program name on the command line.

HLA high-level calling sequence examples:

```
arg.c();
stdout.put( "Number of arguments: ", (type uns32 eax), nl );
```

HLA low-level calling sequence examples:

```
call arg.c;
mov( eax, argCount );
```

### **procedure arg.v( whichArg:dword );**

*arg.v* returns a pointer to the string that corresponds to the specified command-line argument. Argument indexes are in the range 0..`arg.c()`-1. This function will raise an exception if you pass it an argument value outside this range. This function returns a pointer to a string whose storage is internal to the standard library. You must treat this data as read-only data and not modify this data.

HLA high-level calling sequence examples:

```
arg.c();
mov( eax, edx );
for( mov( 0, ecx ); ecx < edx; inc( ecx ) ) do

    arg.v(ecx);
    stdout.put( "arg[", (type uns32 ecx), "]= ", (type string eax), nl );

endfor;
```

HLA low-level calling sequence examples:

```
pushd( 0 );
call arg.v;
mov( eax, firstArg );
```

```
procedure arg.a_v( whichArg:dword );
```

*arg.a\_v* returns a pointer to the string that corresponds to the specified command-line argument. Argument indexes are in the range 0..*arg.c()*-1. This function will raise an exception if you pass it an argument value outside this range. This function returns a pointer to a string it allocates on the heap. The caller must free this storage (via a call to *str.free*) when it is done using the string.

HLA high-level calling sequence examples:

```
arg.c();
mov( eax, edx );
for( mov( 0, ecx ); ecx < edx; inc( ecx ) ) do

    arg.a_v(ecx);
    stdout.put( "arg[" , (type uns32 ecx), "]=", (type string eax), nl );
    str.free( eax );

endfor;
```

HLA low-level calling sequence examples:

```
pushd( 0 );
call arg.a_v;
mov( eax, firstArg );
.
.
.
str.free( firstArg );
```

## 2.4 Deleting Command Line Arguments

The functions in this category delete command-line arguments from the internal array or deallocate all the command-line arguments from the internal array. When an individual command-line parameter is deleted, the indexes of the following command-line arguments are reduced by one (that is, argument *n*+1 becomes argument *n*, argument *n*+2 becomes argument *n*+1, etc., up to the number of command-line arguments). Also note that deleting a command-line argument reduces the value that *arg.c* returns by one.

```
procedure arg.delete( index:uns32 );
```

*arg.delete* removes the specified command-line parameter from the internal argv array. The index parameter must be in the range 0..*argc*, where *argc* is the current value that *arg.c()* returns. If the index parameter is outside this range, this function will raise an *ex.BoundsError* exception. This function decrements the value that *arg.c* returns by one. Note that this function does not affect the value that *arg.cmdLn* returns.

HLA high-level calling sequence examples:

```
arg.c();
stdout.put( "Number of arguments: ", (type uns32 eax), nl );
arg.delete( 0 );
arg.c();
stdout.put
(
    "Number of arguments after delete: ",
    (type uns32 eax),
    nl
);
```

HLA low-level calling sequence examples:

```
pushd( 2 );
call arg.delete;
```

### **procedure arg.destroy();**

*arg.destroy* deletes all of the command line parameters from internal storage and resets the command-line processor. The next function that requests a command line parameter value will force the run-time system to regenerate the command-line argument list.

HLA high-level calling sequence examples:

```
arg.destroy();
arg.c(); // Regenerates original command-line
mov( eax, originalArgc );
```

HLA low-level calling sequence examples:

```
call arg.destroy;
call arg.c;
mov( eax, originalArgc );
```

## **2.5 Argument Iterators**

The iterators in this category process command-line arguments within a foreach loop.

### **iterator arg.args();**

*arg.args* is an iterator (used in an HLA foreach loop) that returns successive command line parameters on each iteration of the foreach loop. This iterator returns a pointer to a newly allocated string in the EAX register. It is the caller's responsibility (usually in the body of the foreach loop) to free the allocated storage with a call to *str.free*.

HLA high-level calling sequence examples:

```
foreach arg.args() do

    stdout.put( "current Arg: ", (type string eax), nl );
    str.free( eax );

endfor;
```

HLA low-level calling sequence examples:

(You really should only use the high-level calling sequence for a foreach loop that calls an iterator.)

### **iterator arg.globalOptions( options:cset );**

*arg.globalOptions* is an iterator (i.e., you use it in a FOREACH loop) that yields a sequence of command line parameter options. A command line parameter option is a command line parameter that begins with a '-' or '/' character. *arg.globalOptions* only returns those command line parameters whose first character is a member of the "options" character set.

A typical command line might be something like the following:



```
c:> pgmName -o2 -warn filename1 -c filename2 -d filename3 -x
```

The command line options in this example are "-o2", "-warn", "-c", "-d", and "-x". The `arg.globalOptions` iterator only considers those command line parameters that begin with "-" and whose first character is a member of the options parameter. Assuming options contains at least {'c', 'd', 'o', 'w', 'x'} then the `arg.globalOptions` iterator will return the strings "o2", "warn", "c", "d", and "x", in that order (that is, in the order they appear on the command line). If the first character of a command line option is not in the options character set, then `arg.globalOptions` does not return that particular command line parameter.

Note that *arg.globalOptions* does not remove the command line parameters from the command line string or from the `arg.v` array. If you want to remove them, you must explicitly do so using the *arg.delete* function. Note, however, that you must not delete command line arguments while scanning through the arguments in a FOREACH loop using the *arg.globalOptions* iterator.

Note that this iterator allocates storage for each string it returns on the heap. It is the caller's responsibility to free the storage when the caller is done using the string.

HLA high-level calling sequence examples:

```
foreach arg.globalOptions( {'a', 'b', 'c' } ) do

    stdout.put( "current option: ", (type string eax), nl );
    str.free( eax );

endfor;
```

HLA low-level calling sequence examples:

```
(You really should only use the high-level calling
sequence for a foreach loop that calls an iterator.)
```

#### **iterator arg.localOptions( options:cset );**

*arg.localOption* is an iterator that yields the sequence of command line options beginning at parameter "index". This iterator only yields strings as long as successive parameters begin with a "-". It fails upon encountering a command line parameter that is not an option (that is, begins with "-"). Note that this iterator only yields those command line options whose first character beyond the "-" character is a member of the options character set.

Typically, you would use the *arg.localOption* iterator inside a FOREACH loop to obtain the command line parameters for a specific filename on the command line. That is, some programs process multiple files and let you associate command line parameters with a single filename. Consider the following simple example:

```
c:> pgm -o2 -c file1 -o5 file2 -c file3
```

In this example, the "pgm" program (presumably) associates "-o2" and "-c" with file1, "-o5" with file2, and "-c" with file3.

Were you to call *arg.localOption* as follows:

```
foreach arg.localOption( 1, {'o', 'c'} ) do ... endfor;
```

then the *arg.localOption* iterator would return two strings: the first would be "o2" and the second would be "c". Within that iterator your code should save these options and count the number of command line parameters processed so it will know the index of the associated filename command line parameter once it is done processing the options. Typically, you would bury this FOREACH loop (with some minor modifications) inside some other loop that processes each filename (or other command line parameter preceded by command line options).

Note that this iterator allocates storage for each string it returns on the heap. It is the caller's responsibility to free the storage when the caller is done using the string.

See the CmdLnDemo.hla file in the Examples directory of the HLA distribution for an example of each of these routines.

HLA high-level calling sequence examples:

```
foreach arg.localOptions( 4, { 'a', 'b', 'c' } ) do

    stdout.put( "current option: ", (type string eax), nl );
    str.free( eax );

endfor;
```

HLA low-level calling sequence examples:

(You really should only use the high-level calling sequence for a foreach loop that calls an iterator.)

## 3 Arrays Module (arrays.hhf)

The HLA Arrays module provides a set of datatypes, macros, and procedures that simplify array access in assembly language (especially multidimensional array access). In addition to supporting standard HLA arrays with static size declarations, the HLA arrays module also supports dynamic arrays that let you specify the array size at run-time.

**Note:** be sure to read the chapter on "Passing Parameters to Standard Library Routines" (parmpassing.rtf) before reading this chapter.

### 3.1 The Arrays Module

To use the array macros in your application, you will need to include one of the following statements at the beginning of your HLA application:

```
#include( "arrays.hhf" )
or
#include( "stdlib.hhf" )
```

### 3.2 Array Data Types

The *array* namespace defines the following useful data types:

```
#macro array.dArray( type, dimensions );
```

The first feature in the array package to consider is the support for dynamic arrays. HLA provides a macro/data type that lets you tell HLA that you want to specify the array size under program control. This macro/data type is *array.dArray* (*dArray* stands for *dynamic array*). You use this macro invocation in place of a standard data type identifier in an HLA variable declaration.

The first macro parameter is the desired data type; this would typically be an HLA primitive data type like *int32* or *char*, although any data type identifier is legal.

The second parameter is the number of dimensions for this array data type. Generally this value is two or greater (since creating dynamic single dimensional arrays using only malloc is trivial). Because of the way array indices are computed by HLA, it is not possible to specify the number of dimensions dynamically.

Note: since *array.dArray* is not a data type identifier (it's a macro), you cannot directly create a dynamic array of dynamic arrays. I.e., the following is not legal:

```
static
  DAofDAs: array.dArray( array.dArray( uns32, 2 ), 2 );
```

However, you can achieve exactly the same thing by using the following code:

```
type
  DAs: array.dArray( uns32, 2 );

static
  DAofDAs: array.dArray( DAs, 2 );
```

The TYPE declaration creates a type identifier that is a dynamic array. The STATIC variable declaration uses this type identifier in the *array.dArray* invocation to create a dynamic array of dynamic arrays.

### 3.3 Array Allocation and Deallocation

```
#macro array.daAlloc( dynamicArrayName, <<list of dimension bounds>> );
```

The *array.dArray* macro allocates storage for a dynamic array variable. It does not, however, allocate storage for the dynamic array itself; that happens at run-time. You must use the *array.daAlloc* macro to actually allocate storage for your array while the program is running. The first parameter must be the name of the dynamic array variable you've declared via the *array.dArray* macro. The remaining parameters are the number of elements for each dimension of the array. This list of dimension bounds must contain the same number of values as specified by the second parameter in the *array.dArray* declaration. The dimension list can be constants or memory locations (note, specifically, that registers are not currently allowed here; this may be fixed in a future version).

The following code demonstrates how to declare a dynamic array and allocate storage for it at run-time:

```
program main;
static
    i:uns32;
    j:uns32;
    k:uns32;
    MyArray: array.dArray( uns32, 3 );

begin main;

    stdout.put( "Enter size of first dimension: " );
    stdin.get( i );
    stdout.put( "Enter size of second dimension: " );
    stdin.get( j );
    stdout.put( "Enter size of third dimension: " );
    stdin.get( k );

    // Allocate storage for the array:

    array.daAlloc( MyArray, i, j, k );

    << Code that manipulates the 3-D dynamic array >>

end main;
```

```
#macro array.daFree( dynamicArrayName );
```

Use the *array.daFree* macro to free up storage you've allocated via the *array.daAlloc* call. This returns the array data storage to the system so it can be reused later. Warning: do not continue to access the array's data after calling *array.daFree*. The system may be using the storage for other purposes after you release the storage back to the system with *array.daFree*.

Note: You should only call *array.daFree* for arrays you've allocated via *array.daAlloc*.

Example:

```
// Allocate storage for the array:

array.daAlloc( MyArray, i, j, k );

<< Code that manipulates the 3-D dynamic array >>

array.daFree( MyArray );
```

## 3.4 Array Predicates

**#macro array.IsItVar( objectName )**

This is a macro that evaluates to a compile-time expression yielding true if the object is a variable identifier. Variable identifiers are those IDs you declare in a VAR, STATIC, READONLY, or STORAGE declaration section, or IDs you declare as parameters to a procedure. This macro returns false for all other parameters.

**#macro array.IsItDynamic( arrayName )**

This is a macro that expands to a compile-time expression yielding true or false depending upon whether the parameter was declared with the *array.dArray* data type. If so, this function returns true; else it returns false. Note that a return value of false does not necessarily indicate that the specified parameter is a static array. *Anything* except a dynamic array object returns false. For example, if you pass the name of a scalar variable, an undefined variable, or something that is not a variable, this macro evaluates false. Note that you can use the HLA *@type* function to test to see if an object is a static array; however, *@type* will not return *hla.ptArray* for dynamic array objects since *array.dArray* objects are actually records. Hence the *array.IsItDynamic* function to handle this chore.

## 3.5 Array Element Access

**#macro array.index( reg32, arrayName, <<list of indices>> );**

This macro computes a row-major order index into a multidimensional array. The array can be a static or dynamic array. The list of indices is a comma separate list of constants, 32-bit memory locations, or 32-bit registers. You should not, however, specify the register appearing as the first parameter in the list of indices.

If the VAL constant *array.BoundsChk* is true, this macro will emit code that checks the bounds of the array indices to ensure that they are valid. The code will raise an *ex.ArrayBounds* exception if any index is out of bounds. You may disable the code generation for the bounds checking by setting the *array.BoundsChk* VAL object to false using a statement like the following:

```
?array.BoundsChk := false;
```

You can turn the bounds checking on and off in segments of your code by using statements like the above that set *array.BoundsChk* to true or false.

This macro leaves pointer into the array sitting in the specified 32-bit register.

Example:

```
static
arrayS: uns32[ 2,3,4 ];
arrayD: array.dArray( uns32, 3 );

.
.
.
// copy arrayS[i, j, k] to arrayD[m,n,p]:

array.index( ebx, arrayS, i, j, k );
mov( [ ebx ], eax );// EAX := arrayS[i,j,k];
array.index( ebx, arrayD, m, n, p );
mov( eax, [ebx] );// EAX := arrayD[m,n,p];
```

**iterator array.element( arrayName );**

This iterator returns each successive element of the specified array. It returns the elements in row major order (that is, the last dimension increments the fastest and the first dimension increments the slowest when returning elements of a multidimensional array). This iterator returns byte objects in the AL register; it returns word objects in the AX register; it returns dword objects in the EAX register; it returns 64-bit (non-real) objects in the EDX:EAX register pair. This routine returns all floating point (real) objects on the top of the FPU stack.

Note that *array.element* is actually a macro, not an iterator. The macro, however, simply provides overloading to call one of seven different iterators depending on the size and type of the operand. However, this macro implementation is transparent to you. You would use this macro exactly like any other iterator.

Note that *array.element* works with both statically declared arrays and dynamic arrays you've declared with *array.dArray* and you've allocated via *array.daAlloc*.

Examples:

```
static
arrayS: uns32[ 2,3,4 ];
arrayD: array.dArray( uns32, 3 );

.
.
.

foreach array.element( arrayS ) do

    stdout.put( "Current arrayS element = ", eax, nl );

endfor;

.
.
.
foreach array.element( arrayD ) do

    stdout.put( "Current arrayD element = ", eax, nl );

endfor;

.
.
.
```

## 3.6 Array Operations

**#macro array.cpy( srcArray, destArray );**

This macro copies a source array to a destination array. Both arrays must be the same size and shape (shape means that they have the same number of dimensions and the bounds on all the dimensions correspond between the source and destination arrays). Both static and dynamic array variables are acceptable for either parameter.

Example:

```
static
arrayS: uns32[ 2,3,4 ];
arrayD: array.dArray( uns32, 3 );

.
.
.

// note: for the following to be legal at run-time,
// the arrayD dynamic array must have storage allocated
// for it with a statement like
//      "array.daAlloc( arrayD, 2, 3, 4 );"

array.cpy( arrayS, arrayD );
```

```
#macro array.reduce( srcArray, destArray );
#keyword array.beforeRow;
#keyword array.reduction;
#keyword array.afterRow;
#terminator array.endreduce;
```

The *array.reduce* macro emits code to do a "row-reduction" on an array. A row reduction is a function that compresses all the rows (that is, the elements selected by running through all the legal values of the last dimension) to a single element. Effectively, this macro reduces an array of arity *n* to an array of arity *n*-1 by eliminating the last dimension.

Reduction is not accomplished by simply throwing away the data in the last dimension (although it's possible to do this). Instead, you've got to supply some code that the *array.reduce* macro will use to compress each row in the array.

A very common reduction function, for example, is addition. Reduction by addition produces a new array that contains the sums of the rows in the previous array. For example, consider the following matrix:

```
1  2  3  4
6  5  4  1
5  9  8  0
```

This is a 3x4 array. Reducing it produces a one dimensional array with three elements containing the value 10, 16, 22 (the sums of each of the above rows).

The best way to understand how the *array.reduce* macro works is to manually implement addition reduction manually. To reduce the 3x4 array above to a single array with three elements, you could use the following code:

```
// (a) Any initialization required before loops
//      (this example requires no such initialization.)

for( mov( 0, i ); i < 3; inc( i ) ) do

    mov( 0, eax );// (b) Initialize sum for each row.

    for( mov( 0, j ); j < 4; inc( j ) ) do

        // (c) Sum up each element in this row into EAX:

        index( ebx, array3x4, i, j );
        add( [ebx], eax );

    endfor;

    // (d) At the end of each row, store the sum away
    // into the destination array.

    mov( i, ebx );
    mov( eax, array3[ ebx*4 ] );

endfor;
```

The *array.reduce* macro is an example of an HLA *context-free macro construct*. This means that the call to *array.reduce* consists of multiple parts, just like the REPEAT..UNTIL and SWITCH..CASE..ENDSWITCH control structures. Specifically, an *array.reduce* invocation consists of the following sequence of macro invocations:

```
array.reduce( srcArray, destArray );

<< Initialization statements needed before
```

```

        loops, (a) in the code above >>

array.beforeRow;

    << Initialization before each row, (b) in the
    code above. Note that edi contains the row
    number times the size of an element and esi contains
    an index into the array to the current element. >>

array.reduction;

    << Code that compresses the data for each
    row, to be executed for each element
    in the row. Corresponds to (c) in the
    the code above. Note that ecx contains
    the index into the current row. >>

array.afterRow;

    << Code to process the compressed data at
    the end of each row. Corresponds to (d)
    in the code above. >>

array.endreduce;

```

A conversion of the previous code to use the `array.reduce` macro set looks like the following:

```

array.reduce( array3x4, array3 )

    // No pre-reduction initialization...

array.beforeRow

    mov( 0, eax );// Initialize the sum for each row.

array.reduction

    add( array3x4[esi], eax );

array.afterRow

    mov( i, edx );
    mov( eax, array3[ edx*4 ] );

array.endreduce;

```

Note that the `array.reduce` macro set makes extensive use of the 80x86 register set. The EAX and EDX registers are the only free registers you can use (without restoring) within the macro. Of course, *array.reduce* will preserve all the registers it uses, but within the macro itself it assumes it can use all registers except EAX and EDX for its own purposes.

**#macro array.transpose( srcArray, destArray, optionalDimension);**

The *array.transpose* macro copies the source array to the destination array transposing the elements of the last dimension with the dimension specified as the last parameter. For the purposes of this macro, the array dimensions of an n-dimensional array are numbered as follows:

```
SomeArray[ n-1, n-2, ..., 3, 2, 1, 0 ];
```



Therefore, *array.transpose* will transpose dimension zero with some other dimension (1..n-1) in the source array when copying the data to the destination array. By default (if you don't supply the optional, third parameter), *array.transpose* will transpose dimensions zero and one when copying the source array to the destination array.

The source and destination arrays must have at least two dimensions. They can be static or dynamic arrays. Note that *array.transpose* emits special, efficient, code when transposing dimensions zero and one.

The source and destination arrays must have compatible shapes. The shapes are compatible if the arrays have the same number of dimensions and all the dimensions have the same bounds except dimension zero and the transpose dimension (which must be swapped). For example, the following two arrays are transpose-compatible when transposing dimensions zero and two:

```
static
  s: uns32[ 2, 2, 3];
  d: uns32[ 3, 2, 2];
```

Generally, one uses *array.transpose* to transpose a two-dimensional matrix. However, the transposition operation is defined for any number of dimensions. To understand how *array.transpose* works, it is instructive to look at the code you'd write to manually transpose the data in an array. Consider the transposition of the data in the *s* and *d* arrays above:

```
for( mov(0, i); i<2; inc(i)) do

  for( mov(0,j); j<2; inc(j)) do

    for( mov(0,k); k<3; inc(k)) do

      index( edx, s, i, j, k );
      mov( [edx], eax );
      index( edx, d, k, j, i );
      mov( eax, [edx] );

    endfor;

  endfor;

endfor;
```

Note that when storing away the value into the destination array, the *i* and *k* indicies were swapped. The following example demonstrates the use of *array.transpose*:

```
static
  s: uns32[2,3] := [1,2,3,4,5,6];
  d: uns32[3,2];
```

```

  .
  .
  .
  array.transpose( s, d );
  .
  .
  .
```

note: The code above copies *s*, as

```
1 2 3
4 5 6
```

to *d*, as

```
1 4
2 5
```

## 3.7 Lookup Tables

The *array.lookupTable* macro lets you easily construct a standard lookup table. This macro invocation must appear within a `STATIC` or a `READONLY` declaration section in your program. A lookup table declaration takes the following form:

```
readonly
  tableName:
    array.lookupTable
    (
      element_data_type,
      default_table_value,
      value: list_of_indexes,
      value: list_of_indexes,
      .
      .
      .
      value: list_of_indexes,
      value: list_of_indexes
    );
```

where:

*element\_data\_type* is the data type for each element of the array, for example, *byte*.

*default\_table\_value* is a value to use for "holes" in the table for which you do not supply an explicit index/value.

*value* is some value that you want to use to initialize a sequence of one or more table entries with.

*list\_of\_indexes* is a list of values that specify indexes into the lookup table. Each entry in a specific list is separated from the other entries with a space (not commas!). Note that each index value you specify must be unique across all lists of indexes in this table (that is, you cannot put two values into the array element specified by a single index). The *array.lookupTable* macro will report an error if you specify a non-unique index value in one of the lists.

Here is a concrete example:

```
static
  tableName:
    array.lookupTable
    (
      int32,
      -1,
      0: 1 2 3 4,
      1: 5 6 7 8,
      2: 9 10 11 12,
      3: 13 14 15,
      4: 20 22 24,
      5: 16 21 25,
      6: 23 19 18
    );
```

This declaration creates a table with 25 dword entries, that will hold the values 1..25. The table will be initialized as follows:

```
tableName:int32[25] :=
[
  0, 0, 0, 0, // Elements 0..3
```

```

1, 1, 1, 1, // Elements 4..7
2, 2, 2, 2, // Elements 8..11
3, 3, 3,    // Elements 12..14
5,          // Element 15
-1,         // Element 16 (no index 17 specified above)
6, 6,       // Elements 17 & 18
4,          // Element 19
5,          // Element 20
4,          // Element 21
6,          // Element 22
4,          // Element 23
5           // Element 24
1;

```

The number of elements appearing in the lookup table will be the difference between the largest index value you supply in all the lists (25 in this example) and the smallest value (1 in this example) plus one. This particular lookup table has 25 entries because  $(25-1+1) = 25$ .

Note that each line in the example above specifies the value to store into each of the table entries in the list that immediately follows. This is probably backwards to what your intuition would suggest. But the nice thing about this arrangement is that it lets you specify a single value to be placed into several different array indices. If there are any gaps in the array indexes you specify (as the value 17 is missing above), then the `array.lookupTable` macro will fill in those entries with the default value specified as the second parameter.

In order to access this lookup table at run-time, you must know the minimum index into the array so you can subtract this from the calculated index you use to access the table. The `array.lookupTable` macro generates four constants for you to help you do this (and other things):

```

tableName_maxValue
tableName_minValue
tableName_maxIndex
tablename_maxIndex

```

The `tableName_minValue` and `tableName_maxValue` constants specify the minimum and maximum index values for the table. In the current example, these constants would be 1 and 25, respectively. The `tableName_minIndex` and `tableName_maxIndex` values are the product of the array element's size with the `_minValue` and `_maxValue` constants. In the table above, the element size is four, so `tableName_minIndex` will be four and `tableName_maxIndex` will be 100. Whenever you access an element of the `tableName` array (in this example), you'll want to subtract the `tableName_minIndex` value from your computed index in order to adjust for non-zero starting indexes, e.g.,

```

mov( someIndex, ebx );
mov( tableName[ebx*4 - tableName_minIndex], eax );

```



## 4 Bit Manipulation (bits.hhf)

The HLA BITS module contains several procedures useful for bit manipulation. Currently, this includes routines like counting bits, reversing bits, and merging bit streams.

**A Note About Thread Safety:** The routines in this module are all thread safe.

**Note about stack diagrams:** this documentation includes stack diagrams for those functions that pass parameters on the stack. To conserve space, this documentation does not include a stack diagram for any function that does not pass data on the stack (that is, only a return address appears on the stack).

### 4.1 Bit Module

To call functions in the Bits module, you must include one of the following statements in your HLA application:

```
#include( "bits.hhf" )
or
#include( "stdlib.hhf" )
```

Note: be sure to read the chapter on "Passing Parameters to Standard Library Routines" (parmpassing.rtf) before reading this chapter.

### 4.2 Bit Counting Function

```
bits.cnt( b:dword in eax ); @returns( "EAX" );
```

This procedure returns the number of one bits present in the "b" parameter (passed in the EAX register). It returns the count in the EAX register. To count the number of zero bits in the parameter value, invert the value of the parameter before passing it to bits.cnt. If you want to count the number of bits in an 8-bit or 16-bit operand, simply zero extend it to 32 bits prior to calling this function. Here are a couple of examples:

If you want to compute the number of bits in an eight-bit operand it's probably faster to write a simple loop that rotates all the bits in the source operand and adds the carry into the accumulating sum. Of course, if performance isn't an issue, you can zero extend the byte to 32 bits and call the bits.cnt procedure.

Note: to count the number of zero bits in an object, first invert than object and then call bits.cnt.

HLA high-level calling sequence examples:

```
bits.cnt( mem32 );
mov( eax, count );

mov( bits.cnt( ebx ), count ); // Note: count is in EAX

// Count the number of bits in 8-bit and 16-bit operands:

bits.cnt( movzx( al, eax ) );
mov( eax, count8 );

mov( bits.cnt( movzx( ax, eax ) ), count16 ); // Count is left in EAX.
```

HLA low-level calling sequence examples:

```
mov( mem32, eax );
call bits.cnt;
mov( eax, count );
```

```

mov( ebx, eax );
call bits.cnt;
mov( eax, count );

movzx( al, eax );
call bits.cnt;
mov( eax, count8 );

movzx( ax, eax );
call bits.cnt;
mov( eax, count16 );

```

## 4.3 Bit Reversal Functions

The functions in this category swap the bits in their input operand. That is, they exchange the H.O. and L.O. bits, the next-to-high-order bit with bit #1, and so on.

**bits.reverse32( b:dword in eax ); @returns( "eax" );**

This function reverses the bits passed to it in EAX and returns the swapped value in EAX. This function swaps bit 31 and 0, bits 30 and 1, bits 29 and 2, bits 28 and 3, and so on. See the diagram for bits.reverse8 and generalize that diagram to 32 bits for a pictorial example.

HLA high-level calling sequence examples:

```

bits.reverse32( mem32 );
mov( eax, reversed32 );

mov( bits.reverse32( ebx ), ebxReversed ); // Note: result is in EAX

```

HLA low-level calling sequence examples:

```

mov( mem32, eax );
call bits.reverse32;
mov( eax, reversed32 );

mov( ebx, eax );
call bits.reverse32;
mov( eax, ebxReversed );

```

**bits.reverse16( b:word in ax ); @returns( "eax" );**

This function reverses the bits passed to it in AX and returns the swapped value in AX. This function swaps bit 15 and 0, bits 14 and 1, bits 13 and 2, bits 12 and 3, and so on. Note that this function does not zero or sign-extend the result into EAX. Although this function currently preserves the H.O. word of EAX, it's safer to assume that the H.O. word of EAX contains an undefined value upon return. See the diagram for bits.reverse8 and generalize that diagram to 16 bits for a pictorial example.

HLA high-level calling sequence examples:

```

bits.reverse16( mem16 );
mov( ax, reversed16 );

```

```
mov( bits.reverse16( bx ), bxReversed ); // Note: result is in AX
```

HLA low-level calling sequence examples:

```
mov( mem16, ax );
call bits.reverse16;
mov( ax, reversed16 );
```

```
mov( bx, ax );
call bits.reverse16;
mov( ax, bxReversed );
```

**bits.reverse8( b:byte in al ); @returns( "eax" );**

This function reverses the bits passed to it in AL and returns the swapped value in AL. This function swaps bit 7 and 0, bits 6 and 1, bits 5 and 2, and bits 4 and 3 (see the diagram for bits.reverse8). Note that this function does not zero or sign-extend the result into EAX. Although this function currently preserves the H.O. three bytes of EAX, it's safer to assume that the H.O. three bytes of EAX contain an undefined value upon return. See the diagram for bits.reverse8 and generalize that diagram to 16 bits for a pictorial example.

HLA high-level calling sequence examples:

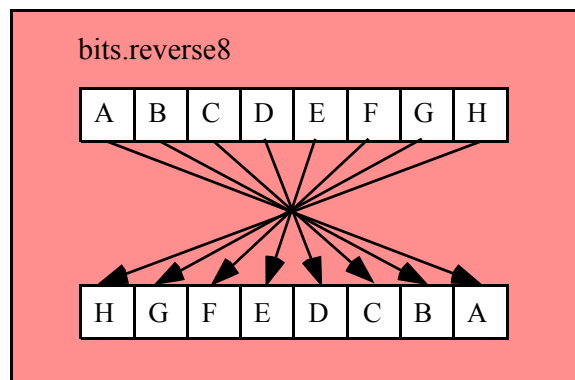
```
bits.reverse8( mem8 );
mov( al, reversed8 );
```

```
mov( bits.reverse8( bl ), blReversed ); // Note: result is in AL
```

HLA low-level calling sequence examples:

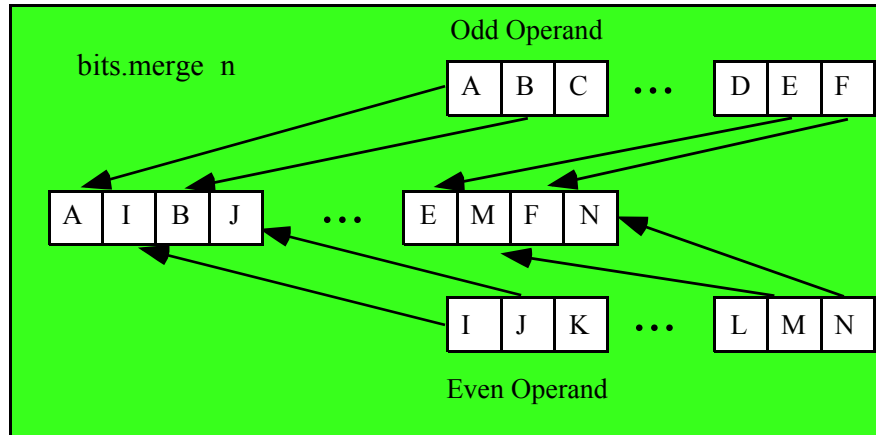
```
mov( mem8, al );
call bits.reverse8;
mov( al, reversed8 );
```

```
mov( bl, al );
call bits.reverse8;
mov( al, blReversed );
```



## 4.4 Bit Merging Functions

The bit merging operands take two small values and produce a single larger value by interleaving the bits from the source operands in the destination operand. One operand's bits are spread out into the destination operand's even bit positions, the other operand's bits are distributed in the odd bit positions (see the following diagram).



```
bits.merge32( even:dword in eax; odd:dword in edx ); @returns( "EDX:EAX" );
```

This function merges two dword values to produce a single qword value. The bits in the even operand are placed in the even bit positions of the qword result, the bits in the odd operand are merged into the odd bit positions. This function returns the qword result in edx:eax (edx contains the H.O. dword).

Note: because this function passes the parameters in EAX and EDX, you may get an undefined result if you specify EDX as the even parameter or EAX as the odd parameter.

HLA high-level calling sequence examples:

```
bits.merge32( mem32Even, mem32Odd );
mov( edx, (type dword mergedQword[4]) );
mov( eax, (type dword mergedQword[0]) );

bits.merge32( ecx, ebx );
mov( edx, (type dword mergedQword[4]) );
mov( eax, (type dword mergedQword[0]) );
```

HLA low-level calling sequence examples:

```
mov( mem32Even, eax );
mov( mem32Odd, edx );
call bits.merge32;
mov( edx, (type dword mergedQword[4]) );
mov( eax, (type dword mergedQword[0]) );

mov( ecx, eax );
mov( ebx, edx );
call bits.merge32;
mov( edx, (type dword mergedQword[4]) );
mov( eax, (type dword mergedQword[0]) );
```



```
bits.merge16( even:word; odd:word ); @returns( "EAX" );
```

This function merges two word values to produce a single dword value. The bits in the even operand are placed in the even bit positions of the dword result, the bits in the odd operand are merged into the odd bit positions. This function returns the dword result in EAX.

HLA high-level calling sequence examples:

```
bits.merge16( mem16Even, mem16Odd );
mov( eax, mergedDword );
```

```
bits.merge16( cx, bx );
mov( eax, mergedDword );
```

HLA low-level calling sequence examples:

```
// If mem16Even and mem16Odd are not at the end
// of some page in memory, you can safely do the following

push( (type dword mem16Even) );
push( (type dword mem16Odd) );
call bits.merge16;
mov( eax, mergedDword );

// To be absolutely safe, do something like the following
// (EAX is free to use because the result comes back in EAX):

movzx( mem16Even, eax );
push( eax );
movzx( mem16Odd, eax );
push( eax );
call bits.merge16;
mov( eax, mergedDword );

push( ecx );
push( edx );
call bits.merge16;
mov( eax, mergedDword );
```

```
bits.merge8( even:byte in al; odd:byte in ah ); @returns( "AX" );
```

This function merges two byte values to produce a single word value. The bits in the even operand are placed in the even bit positions of the word result, the bits in the odd operand are merged into the odd bit positions. This function returns the word result in AX.

Note: because this function passes the parameters in AL and AH, you may get an undefined result if you specify AH as the even parameter or AL as the odd parameter.

HLA high-level calling sequence examples:

```
bits.merge8( mem8Even, mem8Odd );
mov( ax, mergedWord );
```

```
bits.merge8( cl, bh );
mov( eax, mergedWord );
```

HLA low-level calling sequence examples:

```

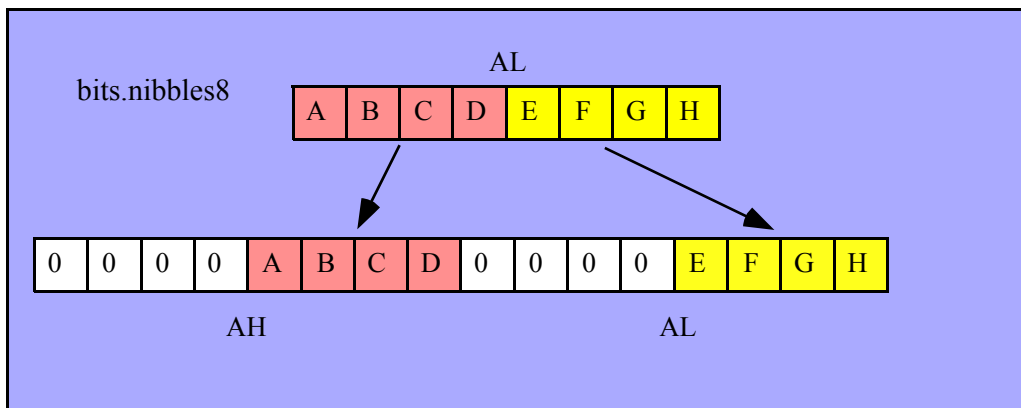
mov( mem8Even, al );
mov( mem8Odd, ah );
call bits.merge8;
mov( ax, mergedWord );

mov( cl, al );
mov( bh, ah );
call bits.merge8;
mov( ax, mergedWord );

```

## 4.5 Bit Extraction Functions

These functions extract nibbles from their source operands, zero extend those nibbles to bytes, and return the values in an operand twice the size of the original operand (e.g., AL->AX, AX->EAX, and EAX->EDX:EAX). For example, the `bits.nibbles8` function does the following:



```
bits.nibbles32( d:dword in eax ); @returns( "EDX:EAX" );
```

This function extracts the 8 nibbles from EAX, zero extends each of them to 8 bits, and then places them in the following locations

```

EAX[0:3]    -> EAX[0:7]
EAX[4:7]    -> EAX[8:15]
EAX[8:11]   -> EAX[16:23]
EAX[12:15]  -> EAX[24:31]
EAX[16:19]  -> EDX[0:7]
EAX[20:23]  -> EDX[8:15]
EAX[24:27]  -> EDX[16:23]
EAX[28:31]  -> EDX[24:31]

```

HLA high-level calling sequence examples:

```

bits.nibbles32( mem32 );
mov( eax, LONibbles );
mov( edx, HONibbles );

bits.nibbles32( ecx );
mov( eax, LONibbles );
mov( edx, HONibbles );

```

HLA low-level calling sequence examples:

```
mov( mem32, eax );
call bits.nibbles32;
mov( ax, mergedWord );

mov( ecx, eax );
call bits.nibbles32;
mov( eax, LONibbles );
mov( edx, HONibbles );
```

**bits.nibbles16( w:word in ax ); @returns( "EAX" );**

This function extracts the 4 nibbles from AX, zero extends each of them to 8 bits, and then places them in the following locations

```
AX[0:3]    -> EAX[0:7]
AX[4:7]    -> EAX[8:15]
AX[8:11]   -> EAX[16:23]
AX[12:15]  -> EAX[24:31]
```

HLA high-level calling sequence examples:

```
bits.nibbles16( mem16 );
mov( eax, Nibbles );

bits.nibbles16( cx );
mov( eax, Nibbles );
```

HLA low-level calling sequence examples:

```
mov( mem16, ax );
call bits.nibbles16;
mov( ax, mergedWord );

mov( cx, ax );
call bits.nibbles16;
mov( eax, LONibbles );
```

**bits.nibbles8( b:byte in al ); @returns( "AX" );**

This function extracts the 2 nibbles from AL, zero extends each of them to 8 bits, and then places them in the following locations

```
AL[0:3] -> AL[0:7]
AL[4:7] -> AH[0:7]
```

HLA high-level calling sequence examples:

```
bits.nibbles8( mem8 );
mov( ax, TwoNibbles );

bits.nibbles8( ch );
mov( ax, TwoNibbles );
```

HLA low-level calling sequence examples:

```
mov( mem8, al );
call bits.nibbles8;
mov( ax, TwoNibbles );

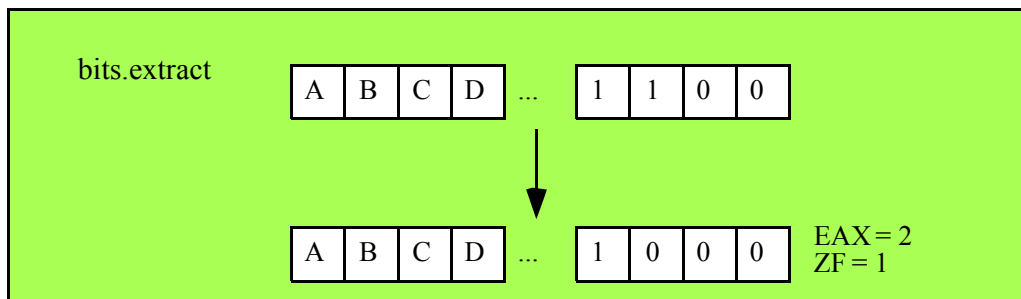
mov( ch, al );
call bits.nibbles8;
mov( ax, TwoNibbles );
```

```
procedure bits.extract( var d:dword );
```

```
  @returns( "EAX" ); // Really a macro.
```

This function extracts the first set bit in *d* searching from bit #0 and returns the index of this bit in the EAX register; the function will also return the zero flag clear in this case. This function also clears that bit in the operand. If *d* contains zero, then this function returns the zero flag set and EAX will contain -1.

Note that HLA actually implements this function as a macro, not a procedure. This means that you can pass any double word operand as a parameter (i.e., a memory or a register operand). However, the results are undefined if you pass EAX as the parameter (since this function returns the bit number in EAX). Note that the macro will report an error message if you try to pass EAX as the parameter.



## 4.6 Bit Distribution Functions

```
bits.distribute( source:dword; mask:dword; dest:dword );
```

```
  @returns( "EAX" );
```

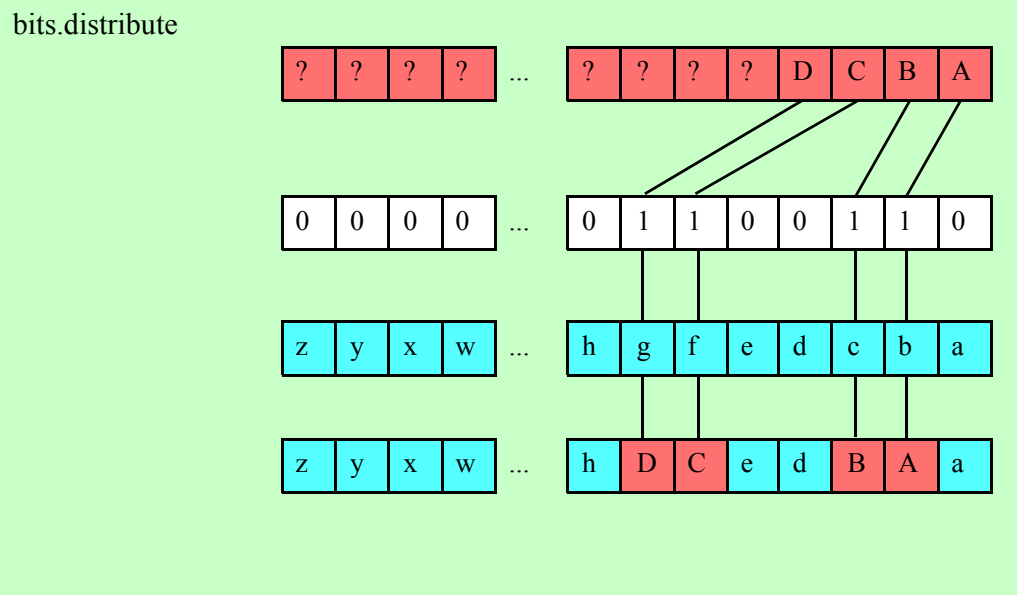
This function takes the L.O. *n* bits of *source*, where *n* is the number of "1" bits in *mask*, and inserts these bits into *dest* at the bit positions specified by the "1" bits in *mask*. This function does not change the bits in *dest* that correspond to the zeros in the *mask* value. This function does not affect the value of the actual *dest* parameter, instead, it returns the new value in the EAX register.

Example:

```
source = $FF00_AA55
mask   = $F0FF_000F
dest   = $1234_5678
```

The `bits.distribute` function grabs the L.O. 16 bits of *source* and inserts these bits at positions 0, 1, 2, 3, 16, 17, 18, 19, 20, 21, 22, 23, and 28, 29, 30, and 31 into the value `$1234_5678` and returns the result in EAX. For this example, `bits.distribute` begins by grabbing the L.O. four bits of *source* and inserts them at bit positions 0..3 of `$1234_5678` (since *mask* contains four set bits at positions 0..3). This yields the temporary result

\$1234\_5675. Next, `bits.distribute` grabs bits 4..11 from source (\$A5) and inserts these bits into bit positions 16..23 since mask contains eight consecutive bits between positions 16 and 23. The produces the temporary result \$12A5\_5675. Finally, `bits.distribute` grabs bits 12..15 from the source operand (\$A) and inserts these into the result at bit positions 28..31 (since the mask value contains set bits at these positions). The final result this function returns in EAX is \$A2A5\_5675.



```
bits.coalesce( source:dword; mask:dword );
```

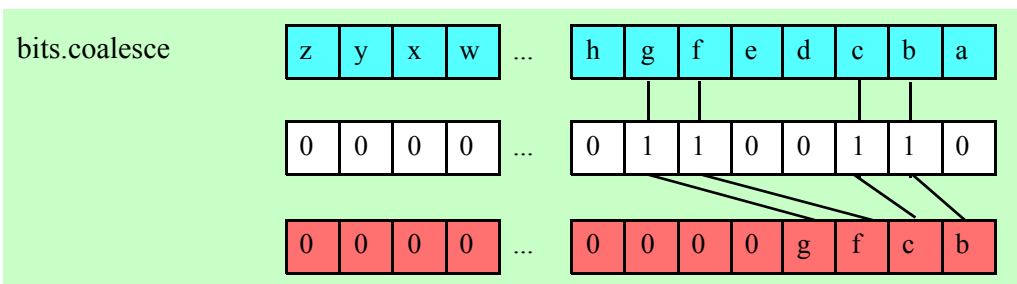
```
@returns( "EAX" );
```

This function is the converse of `bits.distribute`. It extracts all the bits in source whose corresponding positions in mask contain a one. This function coalesces (right justifies) these bits in the L.O. bit positions of the result and returns the result in EAX.

Example:

```
source = $afff_ffce
mask = $aaaa_5555
```

`bits.coalesce` grabs bits 0, 2, 4, 6, 8, 10, 12, 14, 17, 19, 21, 23, 25, 27, 29, and 31 from source and packs these into the L.O. 16 bits of EAX (it also sets the H.O. bits of EAX to zero). The final result in EAX is \$FFFA.





## 5 The Blobs Module (blobs.hhf)

This unit contains routines that read data from and write data to memory-based streams (blobs, or Binary Large Objects). The blob functions can be broken down into five generic categories: general functions that initialize and allocate blobs, functions that convert between blobs and generic memory buffers, functions that do blob file I/O, accessor functions that provide access to the internal blob data structure, and blob I/O functions.

Note: be sure to read the chapter on "Passing Parameters to Standard Library Routines" (parmpassing.rtf) before reading this chapter.

**A Note About the FPU:** The Standard Library code makes occasional use of the FPU, particularly when converting between real and string formats and when computing certain mathematical functions. You should exercise caution when using MMX instructions in a program that makes use of the Standard Library. In particular, you should ensure that you are always in FPU mode (by executing an EMMS instruction) after you are finished using MMX instructions. Better yet, you should avoid the MMX instruction set altogether and use the improved SSE instruction set that accomplishes the same tasks (and doesn't disturb the FPU).

**A note about thread safety:** Each blob object maintains its own critical section variable. Therefore, blobs are protected from multi-threaded access on a per-blob level. Because blobs create this critical section object whenever a blob is initialized or allocated, you must ensure that you free or destroy the blob when you are done using it (and before your program quits) in order to free up system resources.

### 5.1 Conversion Format Control

The blob output functions that convert numeric values to hexadecimal, unsigned decimal, and signed decimal output provide the ability to inject underscores between groups of three (decimal) or four (hexadecimal) digits to make it easier to read large numbers. You enable and disable underscore output using the `conv.setUnderscores` and `conv.getUnderscores` functions. Please refer to their documentation in the `conv.rtf` file for more details.

When reading numeric data from a blob, the blob functions use an internal delimiters character set to determine which characters may legally end a sequence of numeric digits. You can change the complexion of this character set using the `conv.getDelimiters` and `conv.setDelimiters` functions. Please refer to their documentation in the `conv.rtf` file for more details.

When converting numeric values to string form for output, the blob routines call the conversion functions found in the `conv` (conversions) module. For detailed information on the actual conversions, please consult the `conv.rtf` document.

### 5.2 Blob Synopsis

The best way to describe a blob (Binary Large Object) is by calling it a "memory-based file." You generally read and write data to a blob using a file I/O paradigm, that is, by using function calls similar to the HLA standard library's `fileio` functions. The advantage of blobs over files is that blobs are much faster than files; the disadvantage of blobs is that blob data is volatile (meaning that you lose all data written to a blob when the program quits or if there is a power outage). Another pair of blob disadvantages is that blobs' sizes are limited by the amount of available system RAM (files are limited by the amount of free disk space, which is usually much larger) and that you have to predeclare a blob's size before using it. By contrast, files can grow to any size up to the amount of free disk space you having to specify the maximum size prior to creating the file. Despite these limitations, blobs are quite useful because blob manipulation is much faster than file manipulation (often three orders of magnitude faster).

Blobs are very similar to strings. The main differences between strings and blobs are

- Strings generally hold character data (ASCII text), blobs hold arbitrary character and binary data (hence the name "Binary Large Object").
- Strings generally represent a single coherent value (e.g., a single line of text), blobs may contain many records (e.g., multiple lines of text).
- Strings are generally created and manipulated in their entirety; blobs are generally built up and accessed using a sequence of operations (similar to file I/O operations).
- Blobs' data can be aligned on any address that is a power of two (and greater than or equal to 4); strings are always aligned on dword addresses. This makes blobs especially useful for manipulating data objects with certain SSE instructions.

## 5.3 Blob Internal Representation

The following is the internal representation of a blob object (these declarations are inside the blob namespace):

```
type
  blobRec:
    record := -20;

        allocPtr      :dword;
        criticalSection :dword
        rcursor       :dword;
        wcursor       :dword;
        maxlen        :dword;
        length         :dword;
        blobData       :byte[16]; // Minimum is 16 bytes
    endrecord;

  t      :pointer to blobRec; // blob.t outside blob namespace
  blob_t :t;                  // blob.blob_t outside blob namespace
  blob   :t;                  // blob.blob outside blob namespace
```

Note that the fieldnames in blobRec are generally considered private to the blobRec record declaration and are intended for internal use (by the blobs package) only. Applications should only reference or modify these fields using the blob accessor functions (described a little later in this document).

The allocPtr field contains the address of the start of the memory block allocated for the blob. Blobs can be allocated to any address (that is a power of two and greater than or equal to 4). Because the stdlib's mem.alloc function only guarantees 8-byte alignment, certain calls that allocate storage for a blob might need to allocate extra storage in order to align the blob's data (the blobData field) at an appropriate address in memory. This might insert (an arbitrary amount of) padding bytes before the allocPtr field in the blob data structure. The blob data structure needs to save this allocation address (in allocPtr) so that the application can free the storage consumed by the blob object. Much of the time, the allocPtr field will contain it's own address; for unaligned (the most common) blobs, or blobs aligned on 8-byte or less addresses, no extra padding is necessary. If the blob object is allocated in static memory rather than on the heap (or in memory that was not allocated by the blob library functions), the allocPtr field will contain NULL.

The criticalSection field contains the object that controls access to the blob's critical section in multi-threaded applications. Note that this field is still present (though unused) in single-threaded applications. This is a private field and should never be accessed by code outside the blob library.

The rcursor (read cursor) field contains the offset into the blob's data where the next value will be read from the blob using one of the blob.get\* functions or any of the other blob input functions that read data from the read cursor position.

The wcursor (write cursor) field contains the offset into the blob's data where the next value will be written to in the blob using one of the blob.put\* functions or any of the other blob output functions that write data to the write cursor position in the blob.

The maxlen (maximum length) field contains the maximum size of the blob. If an application attempts to read or write data beyond this point in the blob, the library routines will generate an exception.

The length field is the current (dynamic) length of the blob. This field's value is always less than or equal to maxlen's value.

The blobData field is where the blob's data appears in memory. This is an array of bytes containing at least maxlen bytes (it may contain more, depending on the alignment of the blob allocation). This field always contains at least 16 bytes (hence the declared size of 16 in the blobRec record).

## 5.4 Declaring Blob Variables

HLA stdlib blob variables are pointers. When declaring blob objects, you should use the blob.blob, blob.blob\_t, or blob.t data types. These three names are all synonyms, you can use whatever form you choose. Whenever this documentation refers to a "blob variable", it is actually talking about a pointer to a blob object, that is, a variable of type blob.blob, blob.blob\_t, or blob.t. For example:

```
var
  blobVar      : blob.t;
```



```
blobVar2    : blob.blob_t;
blobVar3    : blob.blob;
```

Note that these declarations all reserve exactly four bytes for the blob variables (blobVar1, blobVar2, and blobVar3). Blob variables always hold pointers to the actual blob data. Of course, like strings and other pointers, these declarations do not actually allocate storage for the blob object itself. You will have to call a function such as blob.alloc to allocate the actual storage for a blob object.

You can also use the blob.init function to initialize a block of memory you've allocated for use as a blob.

See the description of this function for more details.

### 5.4.1 Initializing and Allocating Blob Variables

The HLA Blobs module provides several functions that let you allocate storage for a blob on the heap or associate other storage with the blob. Because you must allocate storage for a blob before using it, these functions will generally be the first functions you call before using a blob.

```
blob.init( var b:var; numBytes:dword ); @returns( "eax" );
```

The blob.init routine initializes a block of memory so that it contains a blob object. This function returns a pointer to the blob object in EAX, which you should store into a blob.t variable. The b argument is the address of a block of bytes that you want to use as the blob object. The numBytes argument is the total size of the block of memory pointed at by the b parameter. Note that at least 20 bytes of the object pointed at by b will be used to hold the blob's metadata (internal data structure). That is, numBytes does not specify the maximum size of the blob; the maximum size will actually be slightly smaller than this value. This function will raise an exception if numBytes is too small (insufficient space to allocate the blob metadata and at least 16 bytes of blob data.).

**Warning:** that this function sets the allocPtr field of the blobRec data structure to NULL. If you have allocated this storage on the heap, it is your responsibility to call mem.free with the original allocation address to return the storage to the heap. You must never call blob.free to free the blob storage that you've initialized with a blob.init call.

**Important:** this function creates a new critical section object and initializes the criticalSection field of the blob data object.

You must call blob.destroy when you are done with the blob to delete the critical section object and return its resources to the operating system.

```
blob.init16( var b:var; numBytes:dword ); @returns( "eax" );
```

Just like blob.init except that this function always returns an address that is aligned to a 16-byte boundary. Note that if the memory block pointed at by b isn't at an appropriate address, blob.init16 will use up to 15 bytes at the beginning of this block as padding bytes to guarantee that the address this function returns in eax is aligned to a 16-byte boundary. Therefore, you should ensure that the block of memory whose address you pass is at least 16 bytes larger than you need to ensure you have enough space after padding is removed from the total. The warnings above apply to blob.init16 as well as blob.init.

```
blob.alloc( size:dword ); @returns( "eax" );
```

The blob.alloc routine allocates storage for a blob object that contains at least size bytes of blob data. It initializes the blob header (blobRec) data structure and returns a pointer to the blob in EAX (that you should store into a blob variable). This function will raise an exception if it cannot allocate the specified amount of storage on the heap for the blob object. Note that this function also allocates storage for the blob header and pads the size of the blob (typically to a multiple of 16 bytes) so this call allocates a little bit more storage than size bytes on the heap. Note that blobs created with the blob.alloc function are always aligned on a 16-byte boundary and they always initialize the blobRec allocPtr field with the address of the storage allocated by an (internal) mem.alloc call.

**Important:** this function creates a new critical section object and initializes the criticalSection field of the blob data object.

You must call blob.free when you are done with the blob to delete the critical section object and return its resources to the operating system.

```
blob.allocAligned( size:dword; alignment:dword); @returns( "eax" );
```

The blob.allocAligned routine also allocates storage for a blob object on the heap. The alignment argument is an integer in the range 0..16 specifying the alignment on a 2\*\*alignment byte boundary. That is, the alignment value is interpreted as follows:

alignment	Blob data aligned to
0	16 bytes
1	16 bytes
2	16 bytes
3	16 bytes
4	16 bytes
5	32 bytes
6	64 bytes
7	128 bytes
8	256 bytes
9	512 bytes
10	1024 bytes
11	2048 bytes
12	4096 bytes
13	8192 bytes
14	16384 bytes
15	32768 bytes
16	65536 bytes

Note that the minimum alignment is always 16 bytes because this is the alignment that blob.alloc guarantees. Note that the blob.allocAligned function might have to add as many as 2\*alignment+32 bytes to the size of the storage allocated on the heap in order to guarantee alignment on the desired address boundary. As for the blob.alloc call, this function initializes the internal allocPtr field with the address of the block of storage allocated on the heap.

**Important:** this function creates a new critical section object and initializes the criticalSection field of the blob data object.

You must call blob.free when you are done with the blob to delete the critical section object and return its resources to the operating system.

```
blob.realloc( theBlob:blob.t; size:dword ); @returns( "eax" );
```

The blob.realloc routine resizes an existing blob. The first argument, theBlob, is the address of the blob object to resize and the second argument is the new maximum size of the resultant blob. This function works by allocating a new blob of the specified size, copying the blob data from the original blob to the new blob, and then freeing the storage associated with the original blob if the allocPtr field contains a non-NULL value.

It will also destroy the critical section object held by the original blob.

**Important:** this function creates a new critical section object and initializes the criticalSection field of the blob data object.

You must call blob.free when you are done with the blob to delete the critical section object and return its resources to the operating system.

Note that there is no blob.reallocAligned function call, but you can easily write your own using the following code:

```
blob.allocAligned( newsize, desiredAlignment );
push( eax );
blob.cpy( originalBlob, eax );
blob.free( originalBlob );
```

```
pop( eax );
mov( eax, originalBlob );
```

**blob.free( theBlob:blob.t );**

The blob.free routine frees the storage associated with a blob object if that blob object was allocated on the heap with a call to blob.alloc or blob.allocAligned (specifically, if the allocPtr field in the blobRec data structure contains a non-NULL value). If the allocPtr field contains NULL, then this function returns without raising an exception. If theBlob is NULL, contains a bad address, or if allocPtr isn't pointing at an object on the heap, then this routine will raise an appropriate exception.

This function will also delete the critical section object held by the blob.

**blob.destroy( theBlob:blob.t );**

The blob.destroy routine deletes the critical section held by the blob. If the allocPtr field is non-NULL, this will also deallocate the storage held by the blob. Functionally, blob.free and blob.destroy are equivalent; though blob.free is intended for blobs allocated via some blob.alloc\* or blob.a\_\* function and blob.destroy is intended for blobs initialized via the blob.init\* functions.

## 5.5 Blob Accessor Functions

The following functions in the HLA blobs unit provide access to the blobRec data structure. Programs should use these accessor functions rather than directly accessing the blobRec fields.

**blob.length( b:blob.t ); @returns( "eax" );**

The blob.length routine returns the value of the blobRec.length field, that is, the current size (actual data) of the blob.

HLA high-level calling sequence example:

```
blob.length( b );
mov( eax, blobLength );
```

HLA low-level calling sequence example:

```
push( b );
call blob.length;
mov( eax, blobLength );
```

**blob.setLength( b:blob.t; newLen:dword );**

The blob.setLength routine sets the value of the blobRec.length field to the value specified by the newLen parameter. You should exercise extreme caution when using this function. The blob.setLength function does not check the value of the newLen argument to determine if it is valid. You could supply a value larger than the actual amount of data currently in the blob (meaning you've just added garbage to the end of the blob) or you could even supply a value that is beyond the allocated size of the blob (creating memory access problems down the road).

HLA high-level calling sequence example:

```
blob.setLength( b, 32768 );
```

HLA low-level calling sequence example:

```
push( b );
pushd( 32768 );
```

```
call blob.setLength;
```

```
blob.maxlen( b:blob.t); @returns( "eax" );
```

The blob.maxlen routine returns the value of the blobRec.maxlen field, that is, the current maximum size of the blob.

HLA high-level calling sequence example:

```
blob.maxlen( b );
mov( eax, blobMaxLen );
```

HLA low-level calling sequence example:

```
push( b );
call blob.maxlen;
mov( eax, blobMaxLen );
```

```
blob.setMaxlen( b:blob.t; newLen:dword );
```

The blob.setMaxlen routine sets the value of the blobRec.maxlen field to the value specified by the newLen parameter. You should exercise extreme caution when using this function. The blob.setMaxlen function does not check the value of the newLen argument to determine if it is valid. You should only set the value of this field when allocating storage for a new blob or if you are shrinking the size of a blob in-place. Expanding the maxlen value does not allocate any more storage for the blob and such action will probably lead to a memory fault later on in your program. Note that this function does not change the value of the blobRec.length field, even if the new value for maxlen is less than the existing length.

HLA high-level calling sequence example:

```
blob.setMaxlen( b, 32768 );
```

HLA low-level calling sequence example:

```
push( b );
pushd( 32768 );
call blob.setMaxlen;
```

```
blob.rcursor( b:blob.t); @returns( "eax" );
```

The blob.rcursor routine returns the value of the blobRec.rcursor field in the EAX register.

HLA high-level calling sequence example:

```
blob.rcursor( b );
mov( eax, blobReadPosition );
```

HLA low-level calling sequence example:

```
push( b );
call blob.rcursor;
mov( eax, blobReadPosition );
```

**blob.setCursor( b:blob.t; newCursor:dword );**

The blob.setCursor routine sets the value of the blobRec.rcursor field to the value specified by the newCursor parameter. You should exercise extreme caution when using this function. The blob.setCursor function does not check the value of the newCursor argument to determine if it is valid. If you point the rcursor field beyond the end of the blob's length you could get a memory fault error or read garbage data.

HLA high-level calling sequence example:

```
blob.setCursor( b, 0 );
```

HLA low-level calling sequence example:

```
push( b );
pushd( 0 );
call blob.setCursor;
```

**blob.wcursor( b:blob.t); @returns( "eax" );**

The blob.wcursor routine returns the value of the blobRec.wcursor field in the EAX register.

HLA high-level calling sequence example:

```
blob.wcursor( b );
mov( eax, blobWritePosition );
```

HLA low-level calling sequence example:

```
push( b );
call blob.wcursor;
mov( eax, blobWritePosition );
```

**blob.setwCursor( b:blob.t; newCursor:dword );**

The blob.setwCursor routine sets the value of the blobRec.rcursor field to the value specified by the newCursor parameter. You should exercise extreme caution when using this function. The blob.setwCursor function does not check the value of the newCursor argument to determine if it is valid. If you point the wcursor field beyond the end of the blob's length you could get a memory fault error or leave garbage data in the middle of the blob.

HLA high-level calling sequence example:

```
blob.setwCursor( b, 0 );
```

HLA low-level calling sequence example:

```
push( b );
pushd( 0 );
call blob.setwCursor;
```

**blob.reset;**

The `blob.reset` routine sets the blob's `rcursor`, `wcursor`, and `length` fields to zero. This effectively restores the blob to its original state when it was created. Note that this does not change any actual data appearing in the blob's memory storage, but since `blob.reset` sets the `length` field to zero, this effectively erases any data present in the blob.

HLA high-level calling sequence example:

```
blob.reset();
```

HLA low-level calling sequence example:

```
call blob.reset;
```

**blob.eof( b:blob.t ); @returns( "@c" );**

This function returns true (1) in EAX and the carry flag if the read cursor is at the end of the blob's data. It returns false (0) otherwise. Note that this function actually returns true/false in EAX even though the "returns" value is "@c". It also returns the EOF state in the carry flag (c=1 if EOF, c=0 if not at EOF).

HLA high-level calling sequence example:

```
while( !(blob.eof( blobPointer )) do
```

```
    << something while not at EOF>>
```

```
endwhile;
```

HLA low-level calling sequence example:

```
whileNotEOF:
```

```
    push( blobPointer );
```

```
    call blob.eof;
```

```
    cmp( al, true );
```

```
    jne atEOF;
```

```
    << something while not at EOF>>
```

```
    jmp whileNotEOF;
```

```
atEOF:
```

## 5.6 Blob Assignment Functions

These functions copy blobs and fill a blob with a single byte, word, or double-word value.

**blob.a\_cpy( b:blob.t ); @returns( "eax" );**

This function creates a new blob on the heap that is a copy of the blob pointed at by the `b` parameter. It returns a pointer to the new blob in the EAX register. When you are done using this new blob, you should free the storage associated with it (and delete the criticalsection it creates) by calling the `blob.free` function.

HLA high-level calling sequence example:

```
blob.a_cpy( someBlob );
```

```
mov( eax, newBlob );
```

HLA low-level calling sequence example:

```
push( someBlob );
call blob.a_cpy;
mov( eax, newBlob);
```

**blob.cpy( src:blob.t; dest:blob.t ); @returns( "eax" );**

This function copies the data from the source blob (pointed at by src) to the destination blob (pointed at by dest) and returns a pointer to the destination blob in EAX. This function raises an ex.BlobOverflow exception if the destination blob isn't large enough to hold the data copied from the source blob.

HLA high-level calling sequence example:

```
blob.cpy( someBlob, destBlob );
```

HLA low-level calling sequence example:

```
push( someBlob );
push( destBlob );
call blob.cpy;
```

**blob.fillb( theValue:byte; numBytes:dword; dest:blob );**

This function fills an existing blob with numBytes copies of the byte value theValue. This function resets the rcursor field to zero and the wcursor and length fields to numBytes. This function raises an ex.BlobOverflow exception if the destination blob (pointed at by dest) is not large enough to hold numBytes bytes.

HLA high-level calling sequence example:

```
blob.fillb( 0, 1024, destBlob );
```

HLA low-level calling sequence example:

```
pushd( 0 );
pushd( 1024 );
push( destBlob );
call blob.fillb;
```

**blob.fillw( theValue:word; numWords:dword; dest:blob );**

This function fills an existing blob with numWords copies of the word value theValue. This function resets the rcursor field to zero and the wcursor and length fields to numWords\*2. This function raises an ex.BlobOverflow exception if the destination blob (pointed at by dest) is not large enough to hold numWords\*2 bytes.

HLA high-level calling sequence example:

```
blob.fillw( 1000, 1024, destBlob );
```

HLA low-level calling sequence example:

```
pushd( 1000 );
pushd( 1024 );
push( destBlob );
call blob.fillw;
```

```
blob.filld( theValue:word; numDwords:dword; dest:blob );
```

This function fills an existing blob with numDwords copies of the dword value theValue. This function resets the rcursor field to zero and the wcursor and length fields to numDwords\*4. This function raises an ex.BlobOverflow exception if the destination blob (pointed at by dest) is not large enough to hold numDwords\*4 bytes.

HLA high-level calling sequence example:

```
blob.filld( 1_000_000, 1024, destBlob );
```

HLA low-level calling sequence example:

```
pushd( 1_000_000 );
pushd( 1024 );
push( destBlob );
call blob.filld;
```

## 5.7 Blob Extraction Functions

These functions extract subranges (slices) of a blob.

```
blob.a_subBlob( src:blob; start:dword; len:dword ); @returns( "eax" );
```

This function creates a new blob on the heap and initializes it with data from an existing blob (pointed at by src). The new blob is initialized with len bytes starting at offset start in blob src. This function returns a pointer to the new blob on the heap in the EAX register. It is the callers responsibility to call blob.free to free the storage associated with this new blob (and delete the critical section that this function creates).

HLA high-level calling sequence example:

```
blob.a_subBlob( someBlob, 1024, 256 );
mov( eax, newBlob );
```

HLA low-level calling sequence example:

```
push( someBlob );
pushd( 1024 );
pushd( 256 );
call blob.a_subBlob;
mov( eax, newBlob );
```

```
blob.subBlob( src:blob; start:dword; len:dword; dest:blob.t );
```

This function copies a sequence from a source blob (src) to a destination blob (dest). It copies len bytes starting at offset start in src to dest. This function raises an ex.BlobOverflow exception if d is too small to receive the blob data.

HLA high-level calling sequence example:

```
blob.subBlob( someBlob, 1024, 256, destBlob );
```

HLA low-level calling sequence example:

```
push( someBlob );
pushd( 1024 );
pushd( 256 );
push( destBlob
call blob.subBlob;
```



## 5.8 Blob Comparison Functions

These functions compare two blobs for equality or inequality.

**blob.eq( left:blob; right:blob.t ); @returns( "@c" );**

This function compares two blobs and returns true in the carry flag and the AL register if the two blobs are equal to one another.

HLA high-level calling sequence example:

```
if( blob.eq( someBlob, anotherBlob ) ) then
    // Do something if blobs are equal
endif;
```

HLA low-level calling sequence example:

```
push( someBlob );
push( anotherBlob );
call blob.eq;
jnc blobsNotEqual;

    // Do something if blobs are equal

blobsNotEqual:
```

**blob.ne( left:blob; right:blob.t ); @returns( "@c" );**

This function compares two blobs and returns true in the carry flag and the AL register if the two blobs are not equal to one another.

HLA high-level calling sequence example:

```
if( blob.ne( someBlob, anotherBlob ) ) then
    // Do something if blobs are not equal
endif;
```

HLA low-level calling sequence example:

```
push( someBlob );
push( anotherBlob );
call blob.eq;
jnc blobsEqual;

    // Do something if blobs are not equal

blobsEqual:
```

## 5.9 Blob Scanning Functions

These functions convert memory buffers (ranges of bytes) to blob objects

```

blob.index( src1:blob; src2:blob.t ); @returns( "@c" );
blob.index( src1:blob; offs:dword src2:blob.t ); @returns( "@c" );
blob.index2( src1:blob; src2:blob.t ); @returns( "@c" );
blob.index3( src1:blob; offs:dword src2:blob.t ); @returns( "@c" );
blob.indexStr( src1:blob; src2:string ); @returns( "@c" );
blob.indexStr( src1:blob; offs:dword src2:string ); @returns( "@c" );
blob.indexStr2( src1:blob; src2:string ); @returns( "@c" );
blob.indexStr3( src1:blob; offs:dword src2:string ); @returns( "@c" );

```

These functions search for the presence of a string or blob within some blob. They return the carry flag set if they find the src2 blob or string within the src1 blob and clear if the src2 string or blob is not found within the src1 blob. The variants with the offs parameter begin searching for the blob or string at offset offs within the blob src1. If the carry comes back set (meaning src2 was found), then the EAX register will contain the offset into src1 where src1 is first found. These functions raise an ex.ValueOutOfRange exception if the offs value is greater than the current length of src1.

HLA high-level calling sequence example:

```

if( blob.index( someBlob, subBlob ) ) then
    // Do something if subBlob is a subblob of someBlob
endif;

```

HLA low-level calling sequence example:

```

push( someBlob );
push( subBlob );
call blob.index;
jnc notPresent;

// Do something if subBlob is a subblob of someBlob

notPresent:

```

```

blob.rindex( src1:blob; src2:blob.t ); @returns( "@c" );
blob.rindex( src1:blob; offs:dword src2:blob.t ); @returns( "@c" );
blob.rindex2( src1:blob; src2:blob.t ); @returns( "@c" );
blob.rindex3( src1:blob; offs:dword src2:blob.t ); @returns( "@c" );
blob.rindexStr( src1:blob; src2:string ); @returns( "@c" );
blob.rindexStr( src1:blob; offs:dword src2:string ); @returns( "@c" );
blob.rindexStr2( src1:blob; src2:string ); @returns( "@c" );
blob.rindexStr3( src1:blob; offs:dword src2:string ); @returns( "@c" );

```

These functions search backwards for the presence of a string or blob within some blob starting at the end of that blob. They return the carry flag set if they find the src2 blob or string within the src1 blob and clear if the src2 string or blob is not found within the src1 blob. The variants with the offs parameter begin searching for the blob or string at offset length-offs within the blob src1. If the carry comes back set (meaning src2 was found), then the

EAX register will contain the offset into src1 where src1 is last found. These functions raise an `ex.ValueOutOfRangeException` exception if the `offs` value is greater than the current length of `src1`.

HLA high-level calling sequence example:

```
if( blob.rindex( someBlob, subBlob ) ) then

    // Do something if subBlob is a subblob of someBlob

endif;
```

HLA low-level calling sequence example:

```
push( someBlob );
push( subBlob );
call blob.rindex;
jnc notPresent;

    // Do something if subBlob is a subblob of someBlob

notPresent:
```

```
blob.chpos( src1:blob; src2:char ); @returns( "@c" );
blob.chpos( src1:blob; offs:dword src2:char ); @returns( "@c" );
blob.chpos2( src1:blob; src2:char ); @returns( "@c" );
blob.chpos3( src1:blob; offs:dword src2:char ); @returns( "@c" );
```

These functions search for the presence of a character within some blob. They return the carry flag set if they find the `src2` character within the `src1` blob and clear if the `src2` character is not found within the `src1` blob. The variants with the `offs` parameter begin searching for the character at offset `offs` within the blob `src1`. If the carry comes back set (meaning `src2` was found), then the EAX register will contain the offset into `src1` where `src2` is first found. These functions raise an `ex.ValueOutOfRangeException` exception if the `offs` value is greater than the current length of `src1`.

HLA high-level calling sequence example:

```
if( blob.chpos( someBlob, someChar ) ) then

    // Do something if someChar is within someBlob

endif;
```

HLA low-level calling sequence example:

```
push( someBlob );
mov( someChar, al );
push( eax );
call blob.chpos;
jnc notPresent;

    // Do something if someChar is within someBlob

notPresent:
```

```

blob.rchpos( src1:blob; src2:char ); @returns( "@c" );
blob.rchpos( src1:blob; offs:dword src2:char ); @returns( "@c" );
blob.rchpos2( src1:blob; src2:char ); @returns( "@c" );
blob.rchpos3( src1:blob; offs:dword src2:char ); @returns( "@c" );

```

These functions search for the presence of a character within some blob, searching from the end of the blob. They return the carry flag set if they find the src2 character within the src1 blob and clear if the src2 character is not found within the src1 blob. The variants with the offs parameter begin searching for the character at offset offs within the blob src1. If the carry comes back set (meaning src2 was found), then the EAX register will contain the offset into src1 where src2 is first found. These functions raise an `ex.ValueOutOfRangeException` exception if the offs value is greater than the current length of src1.

HLA high-level calling sequence example:

```

if( blob.rchpos( someBlob, someChar ) ) then

    // Do something if someChar is within someBlob

endif;

```

HLA low-level calling sequence example:

```

push( someBlob );
mov( someChar, al );
push( eax );
call blob.rchpos;
jnc notPresent;

    // Do something if someChar is within someBlob

notPresent:

```

## 5.10 Blob Concatenation Functions

There are two major types of blob concatenation functions. The first group (consisting of the `blob.cat` macro and the `blob.cat2`, `blob.cat3`, and `blob.a_cat` functions) take all the data from one blob and concatenates that data to the end (that is, at the length offset) of a second blob. These functions set the length and `wcursor` fields to point at the end of the new blob and reset the `rcursor` position of the result to zero.

The second group of concatenation functions take data from a string, a buffer, or a blob and append this to the end of some destination blob starting at the `wcursor` position in the destination blob.

```

blob.a_cat( src1:blob; src2:char ); @returns( "@eax" );

```

This function creates a new blob on the heap whose size is equivalent to the current lengths of the src1 and src2 blobs whose pointers are passed as parameters. This function then copies the data from src1 to the new blob and then appends the data from src2 to the end of this. This function returns a pointer to the new blob in EAX and the length and `wcursor` fields are set to the new blob's length and the `rcursor` field is set to zero. Note that it is the caller's responsibility to call `blob.free` in order to return the allocated storage to the heap and delete the newly created critical section object for the blob.

HLA high-level calling sequence example:

```

blob.a_cat( blob1, blob2 );
mov( eax, newBlob );

```

HLA low-level calling sequence example:

```

push( blob1 );
push( blob2 );
call blob.a_cat;
mov( eax, newBlob );

```

**blob.cat( src:blob; dest:blob );**

**blob.cat2( src:blob; dest:blob );**

These functions concatenate the data from the src blob to the end of the data in the dest blob. They raise an ex.BlobOverflow exception if the result will not fit in the destination blob.

HLA high-level calling sequence example:

```
blob.cat2( srcblob, destblob );
```

HLA low-level calling sequence example:

```

push( srcblob );
push( destblob );
call blob.cat2;
mov( eax, newBlob );

```

**blob.cat( src1:blob; src2:blob; dest:blob );**

**blob.cat3( src1:blob; src2:blob; dest:blob );**

These functions concatenate the data from two blobs (src1 and src2) and store the concatenated result into the dest operand. They raise an ex.BlobOverflow exception if the result will not fit in the destination blob.

HLA high-level calling sequence example:

```
blob.cat3( blob1, blob2, dest );
```

HLA low-level calling sequence example:

```

push( blob1 );
push( blob2 );
push( dest );
call blob.cat3;

```

**blob.catsub( src:blob; start:dword; len:dword; dest:blob );**

**blob.catsub4( src:string; start:dword; len:dword; dest:blob );**

**blob.catsub( src2:blob; start:dword; len:dword; src1:string; dest:blob );**

**blob.catsub5( src2:string; start:dword; len:dword; src1:string; dest:blob );**

The blob.catsub functions are actually overloaded procedures that map to the blob.catsub4 and blob.catsub5 functions, depending on the call signature.

These functions concatenate the data from a string (src) or pair of strings (src1 and src2) and store the concatenated result into the dest blob operand. They raise an ex.BlobOverflow exception if the result will not fit in the destination blob. These functions raise an ex.StringIndexError exception if the start index value is greater than the current length of the string.

The first two functions extract a substring of length len, starting at character position start, from src and concatenate this string to the end of the dest blob.

The second two functions copy the src1 string to the blob and then copy the substring of src2 (specified by start and len) to the end of this blob.

All of these functions concatenate their strings to the blob starting at the wcursor position in the blob. They will leave wcursor pointing beyond the data just concatenated and will update the length field of the blob if the concatenated data extends the length. These functions do not affect the rcursor field of the blob.

HLA high-level calling sequence example:

```
blob.catsub4( strVar, 0, 24, dest );
blob.catsub5( strVar2, 0, 24, strVar1, dest );
```

HLA low-level calling sequence example:

```
push( strVar );
pushd( 0 );
pushd( 24 );
push( dest );
call blob.catsub4;

push( strVar2 );
pushd( 0 );
pushd( 24 );
push( strVar1 );
push( dest );
call blob.catsub5;
```

**blob.a\_catsub( src:blob; start:dword; len:dword; dest:blob );**

This function extracts a substring (src, from start of length len) and creates a new blob on the heap from the substring data. This function returns a pointer to the new blob in EAX. This function sets the rcursor field of the blob to zero and the wcursor and length fields of the blob to len. It is the caller's responsibility to free the storage allocated by the function (and the critical section it creates) by calling blob.free when the caller is done with this blob.

HLA high-level calling sequence example:

```
blob.a_catsub( strVar, 0, 24 );
mov( eax, newBlob );
```

HLA low-level calling sequence example:

```
push( strVar );
pushd( 0 );
pushd( 24 );
call blob.a_catsub;
mov( eax, newBlob );
```

**blob.catbuf2( src:buf\_t; dest:blob );**

**blob.catbuf3a( startBuf:dword; endBuf:dword; dest:blob );**

blob.catbuf2 is a synonym for blob.catbuf3a. As it turns out, a buf\_t object passed on the stack is the same as passing the startBuf and endBuf dwords on the stack.

This function concatenates the bytes from a buffer (specified by the src or startBuf/endBuf variables) to the end of an existing blob (dest). This function stores the buffer data starting at the offset specified by the wcursor field of the blob. The bytes from memory locations startBuf to endBuf-1 are concatenated to the blob. If the new wcursor field value is greater than length, this function also extends the value of the length field. This function raises an ex.BlobOverflow exception if the new blob size would be greater than the maxlen field value.

HLA high-level calling sequence example:

```
blob.catbuf3a( startAddress, endAddressPlus1, destBlob );
```

HLA low-level calling sequence example:

```
push( startAddress );
push( endAddressPlus1 );
pushd( destBlob );
call blob.catbuf3a;
```

```
blob.catbuf3b( src2:buf_t; src1:string; dest:blob );
```

```
blob.catbuf4( startBuf:dword; endBuf:dword; strSrc:string; dest:blob );
```

blob.catbuf3b is a synonym for blob.catbuf4. As it turns out, a buf\_t object passed on the stack is the same as passing the startBuf and endBuf dwords on the stack.

This function concatenates the bytes from a buffer (specified by the src2 or startBuf/endBuf variables) to the end of a string (strSrc) and concatenate this data to the end of an existing blob (dest). This function stores the buffer data starting at the offset specified by the wcursor field of the blob. The bytes from the string (strSrc) and then memory locations startBuf to endBuf-1 are concatenated to the blob. If the new wcursor field value is greater than length, this function also extends the value of the length field. This function raises an ex.BlobOverflow exception if the new blob size would be greater than the maxlen field value.

HLA high-level calling sequence example:

```
blob.catbuf4( startAddress, endAddressPlus1, someStr, destBlob );
```

HLA low-level calling sequence example:

```
push( startAddress );
push( endAddressPlus1 );
push( someStr );
pushd( destBlob );
call blob.catbuf4;
```

## 5.11 Blob Conversion Functions

These functions convert memory buffers (ranges of bytes) to blob objects and strings to blob objects.

```
blob.bufToBlob2( buf:@global:buf_t; b:blob.t );
```

```
blob.bufToBlob3( startBuf:dword; endBuf:dword; b:blob.t );
```

These functions convert a

range of bytes (specified by a starting and ending address) into a blob object. The blob data is stored into the (previously allocated) blob object pointed at by the b parameter.

Note that these function names are actually synonyms for the same function. As it turns out, passing a buf\_t object on the stack produces the exact same stack frame as passing a starting and ending buffer address.

The startBuf parameter is the address of the first byte of the memory block to convert; the endBuf parameter supplies the last address of the buffer *plus one*.

The b parameter must point at a previously allocated and initialized blob object. This blob must be large enough (maxlen) to hold the range of bytes specified by the buffer parameter(s).

HLA low-level calling sequence examples:

```
blob.bufToBlob3( startingAddress, endingAddress, blobPtr1 );
blob.bufToBlob3( ebx, ecx, blobPtr2 );
blob.bufToBlob2( buf_t_Variable, blobPtr3 );
```

HLA low-level calling sequence examples:

```

push( startingAddress );
push( endingAddress );
push( blobPtr1 );
call blob.bufToBlob3;

push( ebx );
push( ecx );
push( blobPtr2 );
call blob.a_bufToBlob2;

push( (type dword buf_t_Variable[0]) );
push( (type dword buf_t_Variable[4]) );
push( blobPtr3 );
call blob.a_bufToBlob1;

```

```
blob.a_bufToBlob1( buf:@global:buf_t ); @returns( "@eax" );
```

```
blob.a_bufToBlob2( startBuf:dword; endBuf:dword ); @returns( "@eax" );
```

These functions convert a range of bytes (specified by a starting and ending address) into a blob object. The blob's data is allocated on the heap and these functions return a pointer to the blob data in the EAX register.

Note that these function names are actually synonyms for the same function. As it turns out, passing a `buf_t` object on the stack produces the exact same stack frame as passing a starting and ending buffer address.

The `startBuf` parameter is the address of the first byte of the memory block to convert; the `endBuf` parameter supplies the last address of the buffer *plus one*.

It is the caller's responsibility to call `blob.free` to free up the allocated storage and release the critical section object when the caller is done using the blob these function create.

HLA low-level calling sequence examples:

```

blob.a_bufToBlob2( startingAddress, endingAddress );
mov( eax, blobPtr1 );
blob.a_bufToBlob2( ebx, ecx );
mov( eax, blobPtr2 );
blob.a_bufToBlob1( buf_t_Variable );
mov( eax, blobPtr3 );

```

HLA low-level calling sequence examples:

```

push( startingAddress );
push( endingAddress );
call blob.a_bufToBlob2;
mov( eax, blobPtr1 );

push( ebx );
push( ecx );
call blob.a_bufToBlob2;
mov( eax, blobPtr2 );

push( (type dword buf_t_Variable[0]) );
push( (type dword buf_t_Variable[4]) );
call blob.a_bufToBlob1;
mov( eax, blobPtr3 );

```



```
blob.strToBlob( src:string; dest:blob );
blob.zstrToBlob( src:string; dest:blob );
```

## 5.12 General Blob I/O Functions

Here are the blob file I/O routines provided by the HLA blobs unit:

```
blob.a_load( FileName: string ); @returns( "eax" );
```

The `blob.a_load` routine opens the file by the specified name, allocates sufficient storage to hold all the data in the file, reads the file's data into the blob, and then closes the file. This function returns a pointer to the initialized blob in the EAX register. You should call `blob.free` to return this storage to the heap when you are done using the blob.

This function initializes the read cursor so that it points at the beginning of the blob data read from the file. It initializes the write cursor to point at the end of the blob's data; note, however, that there is no additional space allocated at the end of the blob, so any attempt to write data to the blob (without resetting the write cursor to some other point in the blob) will produce a blob overflow exception.

```
blob.a_loadExtended( FileName: string; extend:dword ); @returns( "eax" );
```

The `blob.a_loadExtended` routine opens the file by the specified name, allocates sufficient storage to hold all the data in the file plus the number of bytes specified by the `extend` argument, reads the file's data into the blob, and then closes the file. This function returns a pointer to the initialized blob in the EAX register. You should call `blob.free` to return this storage to the heap when you are done using the blob.

This function initializes the read cursor so that it points at the beginning of the blob data read from the file. It initializes the write cursor to point at the end of the blob's data. Because the blob's size has been extended by the value of the second parameter in the call, you can write that many additional bytes to the file.

```
blob.load( filename:string; b:blob.t );
```

The `blob.load` routine opens the file by the specified name and reads the file's data into the blob specified by the `b` parameter. This routine raise an exception if there is a problem opening the file (e.g., the file does not exist). If the file is successfully opened, this function will read the file's data into the blob (raising an exception if the file's data is too large to fit in the blob or if there is an error reading the file).

HLA high-level calling sequence examples:

```
blob.load( filenameStr, b );
  blob.a_load( "myfile2.txt" );
  mov( eax, b2 );
  blob.a_loadExtended( "myfile3.txt", 8192 );
  mov( eax, b3 );
```

HLA low-level calling sequence examples:

```
push( filenameStr );
push( b );
call blob.open;

// Note: If you want to use a string literal for the filename, the best
// solution is to create a string object in the readonly section, e.g.,
//
// readonly
//   filenameStr2 :string := "myfile2.txt";
//
// and just use the "filenameStr2" object you've created.

push( filenameStr2 );
```

```
call blob.a_open;
mov( eax, b2 );
```

// You may also do the following if you have a register available:

```
lea( eax, "myfile3.txt" );
push( eax );
pushd( 8192 );
call blob.a_loadExtended;
mov( eax, b3 );
```

**blob.appendFile( filename:string; b:blob.t.blob );**

This function opens a file, reads its data, and appends that data to the end of an existing blob. It raises the `ex.BlobOverflow` exception if the file is too large to append to the end of the blob specified by the `b` parameter. This call sets the write cursor to the end of the file appended to the blob in memory; it does not affect the value of the read cursor.

HLA high-level calling sequence example:

```
blob.appendFile( fileNameStr, b );
```

HLA low-level calling sequence example:

```
push( fileNameStr );
push( b );
call blob.appendFile;
```

**blob.a\_appendFile( filename:string; b:blob.t.blob ); @returns( "eax" );**

This function creates a new blob on the heap that is the size of the data in the `b` blob plus the size of the file specified by `filename`. It copies the data from the `b` blob to the new blob and then reads the file's data and appends that data to the end of the new blob. It returns a pointer to the new blob in the EAX register (the caller should ultimately call `blob.free` to return this storage to the heap). This call sets the write cursor to the end of the file appended to the blob in memory; the value of the read cursor will be the same value found in the `b` blob.

HLA high-level calling sequence example:

```
blob.a_appendFile( fileNameStr, b );
mov( eax, b2 );
```

HLA low-level calling sequence example:

```
push( fileNameStr );
push( b );
call blob.a_appendFile;
mov( eax, b2 );
```

```
blob.a_appendFileExtended( filename:string; b:blob.t.blob; extend:dword )
{@returns( "eax" )};
```

This function allocates storage for a new blob that is the size of the existing blob plus the size of the file and the extend value. It then copies the blob specified by the `b` parameter to the newly allocated blob and appends the file's data to the end of this blob. Finally, it returns a pointer to the new blob in the EAX register. Note that the original blob (specified by the `b` parameter) is unaffected by this operation. This call sets the write cursor to the end of the file appended to the blob in memory; it sets the value of the read cursor to the same value of the original blob (passed in `b`).

HLA high-level calling sequence example:

```
blob.a_appendFileExtend( fileNameStr, b, 16384 );
```

HLA low-level calling sequence example:

```
push( fileNameStr );
push( b );
pushd( 16384 )
call blob.a_appendFileExtend;
```

```
blob.save( filename:string; b:blob.t );
```

This function writes the blob's data (specified by the `b` parameter) to the file specified by the `filename` parameter. This function will overwrite any existing file.

HLA high-level calling sequence examples:

```
blob.save( fileNameStr, blobVar );
```

HLA low-level calling sequence examples:

```
push( fileNameStr );
push( blobVar );
call blob.save;
```

## 5.13 Blob Binary I/O Routines

```
blob.write( b:blob.t; var src:var; len:dword ); @returns( "eax" );
```

This procedure writes the number of bytes specified by the `len` variable to the blob specified by the `b` parameter (at offset `wcursor` in the blob). The bytes starting at the address of the `src` object are written to the blob. No range checking is done on the `src` address value. It is your responsibility to ensure that the buffer contains at least `len` valid data bytes. Note that `src` is an untyped reference parameter. This means that `blob.write` will take the address of whatever object you provide as this parameter (including pointer variables, which may not be what you want). If you want to pass the value of a pointer variable as the buffer address (rather than the address of the pointer variable) when using the high-level style calling syntax, use the `VAL` keyword as a prefix to the parameter (see the following examples). This function returns the number of bytes written to the blob in the EAX register. If this operation writes bytes beyond the previous length of the blob, it will increment the `blobRec.length` field of the blob appropriately.

HLA high-level calling sequence examples:

```

blob.write( blobPointer, buffer, count );

// If bufPtr is a dword object containing the
// address of the buffer whose data you wish to
// write to the blob:

blob.write( blobPointer, val bufPtr, count );

// The following writes the four-byte value of
// the bufPtr variable to the blob (an unusual
// operation):

blob.write( blobPointer, bufPtr, 4 );

```

HLA low-level calling sequence examples:

```

// Assumes buffer is a static object at a fixed
// address in memory:

push( blobPointer );
pushd( &buffer );
push( count );
call blob.write;

// If a 32-bit register is available and buffer
// isn't at a fixed, static, address:

push( blobPointer );
lea( eax, buffer );
push( eax );
push( count );
call blob.write;

// If a 32-bit register is not available and buffer
// isn't at a fixed, static, address:

push( blobPointer );
sub( 4, esp );
push( eax );
lea( eax, buffer );
mov( eax, [esp+4] );
pop( eax );
push( count );
call blob.write;

// If bufPtr points at the buffer to write,
// then use code like this:

push( blobPointer );
push( bufPtr );
push( count );
call blob.write;

// To write the 4 bytes at bufPtr to
// the file (unusual), you could use
// code like this:

push( blobPointer );
lea( eax, bufPtr );
push( eax );

```

```
pushd( 4 );
call blob.write;
```

```
blob.writeAt( b:blob.t; var src:var; index:dword; len:dword );
@returns( "eax" );
```

This procedure writes the number of bytes specified by the len variable to the blob specified by the b parameter at the offset specified by the index parameter. This procedure does not use nor does it modify the blobRec.wcursor value. The bytes starting at the address of the src object are written to the blob. No range checking is done on the src address value. It is your responsibility to ensure that the buffer contains at least len valid data bytes. Note that src is an untyped reference parameter. This means that blob.writeAt will take the address of whatever object you provide as this parameter (including pointer variables, which may not be what you want). If you want to pass the value of a pointer variable as the buffer address (rather than the address of the pointer variable) when using the high-level style calling syntax, use the VAL keyword as a prefix to the parameter (see the following examples). This function returns the number of bytes written to the blob in the EAX register. If the sum of index+len is greater than the previous length of the blob, then this function will extend the length of the blob. If the value of index is greater than the original length of the blob, then this function will return zero in EAX and will not transfer any data to the blob.

HLA high-level calling sequence example:

```
blob.writeAt( blobPointer, buffer, writeOffset, count );
```

HLA low-level calling sequence example:

```
// Assumes buffer is a static object at a fixed
// address in memory:

push( blobPointer );
pushd( &buffer );
push( writeOffset );
push( count );
call blob.writeAt;
```

```
blob.putByte( b:blob.t; byteVal:byte );
```

This procedure writes a single byte value (byteVal, which is a single-byte binary value) to the blob specified by the b parameter (at offset wcursor in the blob). This function call advances the value of wcursor by 1. This call is effectively equivalent to blob.write( b, byteVal, 1); except that it does not return the number of bytes written in EAX (which is always 1, assuming there are no exceptions).

HLA high-level calling sequence example:

```
blob.putByte( blobPointer, ByteValue );
```

HLA low-level calling sequence example:

```
// Assumes buffer is a static object at a fixed
// address in memory and EAX is available for use:

push( blobPointer );
movzx( ByteValue, eax );
```

```
push( eax );
call blob.putByte;
```

### **blob.putWord( b:blob.t; wordVal:word );**

This procedure writes a single word value (wordVal, which is a two-byte binary value) to the blob specified by the b parameter (at offset wcursor in the blob). This function call advances the value of wcursor by 2. This call is effectively equivalent to blob.write( b, wordVal, 2); except that it does not return the number of bytes written in EAX (which is always 2, assuming there are no exceptions).

HLA high-level calling sequence example:

```
blob.putWord( blobPointer, WordValue );
```

HLA low-level calling sequence example:

```
// Assumes buffer is a static object at a fixed
// address in memory:
```

```
push( blobPointer );
pushw( 0 );
push( WordValue );
call blob.putWord;
```

### **blob.putDword( b:blob.t; dwordVal:dword );**

This procedure writes a single dword value (dwordVal, which is a four-byte binary value) to the blob specified by the b parameter (at offset wcursor in the blob). This function call advances the value of wcursor by 4. This call is effectively equivalent to blob.write( b, dwordVal, 4); except that it does not return the number of bytes written in EAX (which is always 4, assuming there are no exceptions).

HLA high-level calling sequence example:

```
blob.putDword( blobPointer, DwordValue );
```

HLA low-level calling sequence example:

```
// Assumes buffer is a static object at a fixed
// address in memory:
```

```
push( blobPointer );
push( DwordValue );
call blob.putDword;
```

### **blob.putQword( b:blob.t; QwordVal:qword );**

This procedure writes a single qword value (qwordVal, which is an eight-byte binary value) to the blob specified by the b parameter (at offset wcursor in the blob). This function call advances the value of wcursor by 8. This call is effectively equivalent to blob.write( b, qwordVal, 8); except that it does not return the number of bytes written in EAX (which is always 8, assuming there are no exceptions).

HLA high-level calling sequence example:

```
blob.putQword( blobPointer, QwordValue );
```

HLA low-level calling sequence example:

```
// Assumes buffer is a static object at a fixed
// address in memory:

push( blobPointer );
push( (type dword QwordValue[4]) );
push( (type dword QwordValue[0]) );
call blob.putQword;
```

**blob.putTbyte( b:blob.t; tbyteVal:tbyte );**

This procedure writes a single tbyte value (tbyteVal, which is a 10-byte binary value) to the blob specified by the b parameter (at offset wcursor in the blob). This function call advances the value of wcursor by 10. This call is effectively equivalent to blob.write( b, tbyteVal, 10); except that it does not return the number of bytes written in EAX (which is always 10, assuming there are no exceptions).

HLA high-level calling sequence example:

```
blob.putTbyte( blobPointer, TByteValue );
```

HLA low-level calling sequence example:

```
// Assumes buffer is a static object at a fixed
// address in memory:

push( blobPointer );
pushw( 0 );
push( (type word TByteValue [8]) );
push( (type dword TByteValue [4]) );
push( (type dword TByteValue [0]) );
call blob.putTbyte;
```

**blob.putLword( b:blob.t; LwordVal:lword );**

This procedure writes a single lword value (lwordVal, which is a 16-byte binary value) to the blob specified by the b parameter (at offset wcursor in the blob). This function call advances the value of wcursor by 16. This call is effectively equivalent to blob.write( b, lwordVal, 16); except that it does not return the number of bytes written in EAX (which is always 16, assuming there are no exceptions).

HLA high-level calling sequence example:

```
blob.putLword( blobPointer, LwordValue );
```

HLA low-level calling sequence example:

```
// Assumes buffer is a static object at a fixed
// address in memory:

push( blobPointer );
push( (type dword LwordValue[12]) );
push( (type dword LwordValue[8 ] ) );
push( (type dword LwordValue[4 ] ) );
push( (type dword LwordValue[0 ] ) );
```

```
call blob.putLword;
```

```
blob.read( b:blob.t; var buffer:byte; count:uns32 ); @returns( "eax" );
```

This routine reads a sequence of count bytes from the specified blob starting at the rcursor position in the blob. It stores the bytes into memory at the address specified by buffer.

It returns the number of bytes actually read from the blob in the EAX register (this is usually equal to the count value, unless the read operation attempts to read beyond the current length of the blob, in which case the actual number of bytes is returned in EAX).

HLA high-level calling sequence examples:

```
blob.read( blobPointer, buffer, count );
blob.read( blobPointer, [eax], 1024 );
```

HLA low-level calling sequence examples:

```
// If buffer is a static variable:
```

```
push( blobPointer );
pushd( &buffer );
push( count );
call blob.read;
```

```
blob.readAt( b:blob.t; var buffer:byte; index:dword; len:uns32 )
```

This routine reads a sequence of len bytes from the specified blob starting at offset index into the blob. It stores the bytes into memory at the address specified by buffer.

This function call ignores the initial value in the rcursor variable and it does not change this value. This function returns the actual number of bytes read in the EAX register (which is usually equal to len). If len plus index is greater than the current blob length, this this function returns the actual number of bytes read (which will be less than len) in the EAX register.

HLA high-level calling sequence examples:

```
blob.readAt( blobPointer, buffer, index, count );
blob.readAt( blobPointer, [eax], 500, 1024 );
```

HLA low-level calling sequence examples:

```
push( blobPointer );
pushd( &buffer );
push( index );
push( count );
call blob.readAt;
```

```
blob.getBytes( b:blob.t ); @returns( "al" );
```

This procedure reads a single byte value from the blob specified by the b parameter (at offset rcursor in the blob) and returns this byte in the AL register. This function call advances the value of rcursor by 1. This call is effectively equivalent to blob.read( b, byteVal, 1); except that it does not return the number of bytes read in EAX (which is always 1, assuming there are no exceptions).

It will raise an ex.BlobOverflow exception if the value of rcursor is greater than or equal to the current blob length.



HLA high-level calling sequence example:

```
blob.getBytes( blobPointer );
mov( al, ByteValue );
```

HLA low-level calling sequence example:

```
push( blobPointer );
call blob.getBytes;
mov( al, ByteValue );
```

**blob.getWorld( b:blob.t ); @returns( "ax" );**

This procedure reads a single word value from the blob specified by the *b* parameter (at offset *rcursor* in the blob) and returns this word in the AX register. This function call advances the value of *rcursor* by 2. This call is effectively equivalent to `blob.read( b, wordVal, 2)`; except that it does not return the number of bytes read in EAX (which is always 2, assuming there are no exceptions).

It will raise an `ex.BlobOverflow` exception if the value of *rcursor* is greater than or equal to the current blob length.

HLA high-level calling sequence example:

```
blob.getWorld( blobPointer );
mov( ax, WordValue );
```

HLA low-level calling sequence example:

```
push( blobPointer );
call blob.getWorld;
mov( ax, WordValue );
```

**blob.getDword( b:blob.t ); @returns( "eax" );**

This procedure reads a single dword value from the blob specified by the *b* parameter (at offset *rcursor* in the blob) and returns this dword in the EAX register. This function call advances the value of *rcursor* by 4. This call is effectively equivalent to `blob.read( b, dwordVal, 4)`; except that it does not return the number of bytes read in EAX (which is always 4, assuming there are no exceptions).

It will raise an `ex.BlobOverflow` exception if the value of *rcursor* is greater than or equal to the current blob length.

HLA high-level calling sequence example:

```
blob.getDword( blobPointer );
mov( eax, DwordValue );
```

HLA low-level calling sequence example:

```
push( blobPointer );
call blob.getDword;
mov( ax, DwordValue );
```

```
blob.getQword( b:blob.t ); @returns( "edx:eax" );
```

This procedure reads a single qword value from the blob specified by the `b` parameter (at offset `rcursor` in the blob) and returns this qword in the EDX:EAX register pair. This function call advances the value of `rcursor` by 8. This call is effectively equivalent to `blob.read( b, qwordVal, 8)`; except that it does not return the number of bytes read in EAX (which is always 8, assuming there are no exceptions).

It will raise an `ex.BlobOverflow` exception if the value of `rcursor` is greater than or equal to the current blob length.

HLA high-level calling sequence example:

```
blob.getQword( blobPointer );
mov( eax, (type dword QwordValue[0]) );
mov( edx, (type dword QwordValue[4]) );
```

HLA low-level calling sequence example:

```
push( blobPointer );
call blob.getQword;
mov( eax, (type dword QwordValue[0]) );
mov( edx, (type dword QwordValue[4]) );
```

```
blob.getTbyte( b:blob.t; tbyteVal:tbyte );
```

This procedure reads a single tbyte value from the blob specified by the `b` parameter (at offset `rcursor` in the blob) and stores this tbyte via the `tbyteVal` reference parameter. This function call advances the value of `rcursor` by 10. This call is effectively equivalent to `blob.read( b, tbyteVal, 10)`; except that it does not return the number of bytes read in EAX (which is always 10, assuming there are no exceptions).

It will raise an `ex.BlobOverflow` exception if the value of `rcursor` is greater than or equal to the current blob length.

HLA high-level calling sequence example:

```
blob.getTbyte( blobPointer, tByteVar );
```

HLA low-level calling sequence example:

```
push( blobPointer );
pushd( &tbyteVar );
call blob.getTbyte;
```

```
blob.getLword( b:blob.t; lwordVal:lword );
```

This procedure reads a single lword value from the blob specified by the `b` parameter (at offset `rcursor` in the blob) and stores this lword via the `lwordVal` reference parameter. This function call advances the value of `rcursor` by 16. This call is effectively equivalent to `blob.read( b, lwordVal, 16)`; except that it does not return the number of bytes read in EAX (which is always 16, assuming there are no exceptions).

It will raise an `ex.BlobOverflow` exception if the value of `rcursor` is greater than or equal to the current blob length.

HLA high-level calling sequence example:

```
blob.getLword( blobPointer, lwordVar );
```

HLA low-level calling sequence example:

```

push( blobPointer );
pushd( &lwordVar );
call blob.getLword;

```

## 5.14 Blob Output Routines

The output routines in the blobs module are very similar to the file output routines in the fileio module as well as the output routines in the stdout library module. In general, these routines require (at least) two parameters; the first is the (pointer to the) blob object, the second parameter is usually the value to write to the blob. Some functions contain additional parameters that provide formatting information.

All output is written to the blob at the wcursor offset into the blob's data. For each byte written, wcursor is incremented by 1. If wcursor's value becomes greater than or equal to the blob's length value, then length is adjusted as well (that is, length and wcursor will have the same value). If wcursor exceeds the value in the maxlen field, then these functions raise an ex.BlobOverflow exception.

See the descriptions of the corresponding functions in the fileio module for more details.

Note that function names of the form blob.cat\* are synonyms for the blob.put\* functions.

```

blob.newln( b:blob.t )
blob.putbool( b:blob.t; b:boolean )
blob.putc( b:blob.t; c:char )
blob.putcSize( b:blob.t; c:char; width:int32; fill:char )
blob.putcset( b:blob.t; cst:cset )
blob.puts( b:blob.t; s:string )
blob.putsSize( b:blob.t; s:string; width:int32; fill:char )
blob.putb( b:blob.t; b:byte )
blob.puth8( b:blob.t; b:byte )
blob.puth8Size( b:blob.t; b:byte; size:dword; fill:char )
blob.putw( b:blob.t; w:word )
blob.puth16( b:blob.t; w:word )
blob.puth16Size( b:blob.t; w:word; size:dword; fill:char )
blob.putd( b:blob.t; d:dword )
blob.puth32( b:blob.t; d:dword )
blob.puth32Size( b:blob.t; d:dword; size:dword; fill:char )
blob.putq( b:blob.t; q:qword )
blob.puth64( b:blob.t; q:qword )
blob.puth64Size( b:blob.t; q:qword; size:dword; fill:char )
blob.puttb( b:blob.t; tb:tbyte )
blob.puth80( b:blob.t; tb:tbyte )
blob.puth80Size( b:blob.t; tb:tbyte; size:dword; fill:char )
blob.putl( b:blob.t; l:lword )
blob.puth128( b:blob.t; l:lword )
blob.puti8( b:blob.t; b:byte )
blob.puti8Size( b:blob.t; b:byte; width:int32; fill:char )
blob.puti16( b:blob.t; w:word )
blob.puti16Size( b:blob.t; w:word; width:int32; fill:char )
blob.puti32( b:blob.t; d:dword )
blob.puti32Size( b:blob.t; d:dword; width:int32; fill:char )
blob.puti64( b:blob.t; q:qword )
blob.puti64Size( b:blob.t; q:qword; width:int32; fill:char )
blob.puti128( b:blob.t; l:lword )
blob.puti128Size( b:blob.t; l:lword; width:int32; fill:char )
blob.putu8( b:blob.t; b:byte )
blob.putu8Size( b:blob.t; b:byte; width:int32; fill:char )
blob.putu16( b:blob.t; w:word )
blob.putu16Size( b:blob.t; w:word; width:int32; fill:char )
blob.putu32( b:blob.t; d:dword )
blob.putu32Size( b:blob.t; d:dword; width:int32; fill:char )
blob.putu64( b:blob.t; q:qword )

```

```

blob.putu64Size( b:blob.t; q:qword; width:int32; fill:char )
blob.putu128( b:blob.t; l:lword )
blob.putu128Size( b:blob.t; l:lword; width:int32; fill:char )
blob.pute32( b:blob.t; r:real32; width:uns32 )
blob.pute64( b:blob.t; r:real64; width:uns32 )
blob.pute80( b:blob.t; r:real80; width:uns32 )
blob.putr32( b:blob.t; r:real32; width:uns32; decpts:uns32; pad:char )
blob.putr64( b:blob.t; r:real64; width:uns32; decpts:uns32; pad:char )
blob.putr80( b:blob.t; r:real80; width:uns32; decpts:uns32; pad:char )
blob.put( list_of_items )

```

## 5.15 Blob Input Routines

The input routines in the blobs module are very similar to the file input routines in the fileio module as well as the input routines in the stdin library module. In general, these routines require one parameter: the pointer to the blob object.

All input is read from the blob starting at the rcursor offset into the blob's data. For each byte read, rcursor is incremented by 1. If rcursor's value becomes greater than or equal to the blob's length value, then these functions raise an `ex.EndOfFile` exception.

See the descriptions of the corresponding functions in the fileio module for more details.

```

blob.readLn( b:blob.t );
blob.eoln( b:blob.t ); @returns( "al" );
blob.getc( b:blob.t ); @returns( "al" );
blob.gets( b:blob.t; s:string );
blob.a_gets( b:blob.t ); @returns( "eax" );
blob.geth8( b:blob.t ); @returns( "al" );
blob.geth16( b:blob.t ); @returns( "ax" );
blob.geth32( b:blob.t ); @returns( "eax" );
blob.geth64( b:blob.t );@returns( "edx:eax" );
blob.geth80( b:blob.t; var dest:tbyte );
blob.geth128( b:blob.t; var dest:lword );
blob.geti8( b:blob.t ); @returns( "al" );
blob.geti16( b:blob.t ); @returns( "ax" );
blob.geti32( b:blob.t ); @returns( "eax" );
blob.geti64( b:blob.t );@returns( "edx:eax" );
blob.geti128( b:blob.t; var dest:lword );
blob.getu8( b:blob.t ); @returns( "al" );
blob.getu16( b:blob.t ); @returns( "ax" );
blob.getu32( b:blob.t ); @returns( "eax" );
blob.getu64( b:blob.t );@returns( "edx:eax" );
blob.getu128( b:blob.t; var dest:lword );
blob.getf( b:blob.t ); @returns( "st0" );
blob.get( List_of_items_to_read );

```

## 6 Character Classification and Utilities Module (chars.hhf)

The HLA CHARS module contains several procedures that classify and convert various character subtypes. Conversion routines include upper and lower case conversion. Classification routines include checking for alphabetic characters, numeric characters, whitespace characters, etc. This module works with ASCII characters in the range #0..#\$7F only. Though the functions accept 8-bit character values, non-ASCII characters generally do not get translated by the conversion routines and the predicate routines almost always return false for non-ASCII characters.

**A Note About Thread Safety:** The routines in this module are all thread safe.

**Note about stack diagrams:** To conserve space, this documentation does not include a stack diagram for any functions because none of the current "chars" functions pass data on the stack (that is, only a return address appears on the stack).

### 6.1 Conversion Functions

The conversion functions in the chars module convert (alphabetic) characters to lowercase and to uppercase.

```
chars.toUpper( c:byte ); @returns( "AL" );
```

This routine returns the character passed as a parameter in the AL register. If the character passed as a parameter was a lower case alphabetic character, this procedure converts it to upper case before returning it. Character values in the range #128..#255 are returned as-is; no conversion is done on those characters even if in some language they could be interpreted as alphabetic characters.

HLA high-level calling sequence examples:

```
chars.toUpper( charVar );
mov( al, uppercaseCharVar );
```

```
chars.toUpper( al );
// AL now contains the uppercase version.
```

```
mov( chars.toUpper( ch ), uppercaseCharVar ); // Char is left in AL.
```

HLA low-level calling sequence examples:

```
mov( charVar, al );
call chars.toUpper;
mov( al, uppercaseCharVar );
```

```
// Char to convert is in AL
```

```
call chars.toUpper;
// Result char is left in AL
```

```
mov( ch, al );
call chars.toUpper;
mov( al, uppercaseCharVar );
```

```
chars.toLower( c:byte ); @returns( "AL" );
```

This routine returns the character passed as a parameter in the AL register. If the character passed as a parameter was an upper case alphabetic character, this procedure converts it to lower case before returning it.

Character values in the range #128..#255 are returned as-is; no conversion is done on those characters even if in some language they could be interpreted as alphabetic characters.

HLA high-level calling sequence examples:

```
chars.toLower( charVar );
mov( al, lowercaseCharVar );
```

```
chars.toLower( al );
// AL now contains the lowercase version.
```

```
mov( chars.toLower( ch ), lowercaseCharVar ); // Char is left in AL.
```

HLA low-level calling sequence examples:

```
mov( charVar, al );
call chars.toLower;
mov( al, lowercaseCharVar );
```

```
// Char to convert is in AL
```

```
call chars.toLower;
// Result char is left in AL
```

```
mov( ch, al );
call chars.toLower;
mov( al, lowercaseCharVar );
```

## 6.2 Predicates (Tests)

The following functions test characters in the seven-bit ASCII character set. These functions produce undefined results for other character sets. Note: Although the "returns" value for each of these functions is "AL", in reality these functions all return the Boolean result zero-extended to 32 bits in EAX.

**chars.isAlpha( c:byte ); @returns( "AL" );**

This routine returns true in the AL register if the parameter is an alphabetic character.

HLA high-level calling sequence examples:

```
chars.isAlpha( charVar );
mov( al, booleanVar );
```

```
chars.isAlpha( al );
// AL now contains the Boolean result.
```

```
mov( chars.isAlpha( ch ), booleanVar ); // Result is left in AL.
```

HLA low-level calling sequence examples:

```
mov( charVar, al );
call chars.isAlpha;
mov( al, booleanVar );
```

```
// Char to convert is in AL

call chars.isAlpha;
// Result boolean is left in AL

mov( ch, al );
call chars.isAlpha;
mov( al, booleanVar );
```

**chars.isUpper( c:byte ); @returns( "AL" );**

This routine returns true in the AL register if the parameter is an upper case alphabetic character.

HLA high-level calling sequence examples:

```
chars.isUpper( charVar );
mov( al, booleanVar );

chars.isUpper( al );
// AL now contains the Boolean result.

mov( chars.isUpper( ch ), booleanVar ); // Result is left in AL.
```

HLA low-level calling sequence examples:

```
mov( charVar, al );
call chars.isUpper;
mov( al, booleanVar );

// Char to convert is in AL

call chars.isUpper;
// Result boolean is left in AL

mov( ch, al );
call chars.isUpper;
mov( al, booleanVar );
```

**chars.isLower( c:byte ); @returns( "AL" );**

This routine returns true in the AL register if the parameter is a lower case alphabetic character.

HLA high-level calling sequence examples:

```
chars.isLower( charVar );
mov( al, booleanVar );

chars.isLower( al );
// AL now contains the Boolean result.

mov( chars.isLower( ch ), booleanVar ); // Result is left in AL.
```

HLA low-level calling sequence examples:

```
mov( charVar, al );
call chars.isLower;
mov( al, booleanVar );

// Char to convert is in AL

call chars.isLower;
// Result boolean is left in AL

mov( ch, al );
call chars.isLower;
mov( al, booleanVar );
```

**chars.isAlphaNum( c:byte ); @returns( "AL" );**

This routine returns true in the AL register if the parameter is an alphanumeric character.

HLA high-level calling sequence examples:

```
chars.isAlphaNum( charVar );
mov( al, booleanVar );

chars.isAlphaNum( al );
// AL now contains the Boolean result.

mov( chars.isAlphaNum( ch ), booleanVar ); // Result is left in AL.
```

HLA low-level calling sequence examples:

```
mov( charVar, al );
call chars.isAlphaNum;
mov( al, booleanVar );

// Char to convert is in AL

call chars.isAlphaNum;
// Result boolean is left in AL

mov( ch, al );
call chars.isAlphaNum;
mov( al, booleanVar );
```

**chars.isDigit( c:byte ); @returns( "AL" );**

This routine returns true in the AL register if the parameter is a decimal digit character.

HLA high-level calling sequence examples:

```
chars.isDigit( charVar );
mov( al, booleanVar );
```



```

chars.isDigit( al );
// AL now contains the Boolean result.

mov( chars.isDigit( ch ), booleanVar ); // Result is left in AL.

```

HLA low-level calling sequence examples:

```

mov( charVar, al );
call chars.isDigit;
mov( al, booleanVar );

// Char to convert is in AL

call chars.isDigit;
// Result boolean is left in AL

mov( ch, al );
call chars.isDigit;
mov( al, booleanVar );

```

**chars.isXDigit( c:byte ); @returns( "AL" );**

This routine returns true in the AL register if the parameter is a hexadecimal digit character.

HLA high-level calling sequence examples:

```

chars.isXDigit( charVar );
mov( al, booleanVar );

chars.isXDigit( al );
// AL now contains the Boolean result.

mov( chars.isXDigit( ch ), booleanVar ); // Result is left in AL.

```

HLA low-level calling sequence examples:

```

mov( charVar, al );
call chars.isXDigit;
mov( al, booleanVar );

// Char to convert is in AL

call chars.isXDigit;
// Result boolean is left in AL

mov( ch, al );
call chars.isXDigit;
mov( al, booleanVar );

```

**chars.isGraphic( c:byte ); @returns( "AL" );**

This routine returns true in the AL register if the parameter is a printable character or a space (this excludes control characters; also, this function only applies to ASCII characters).

HLA high-level calling sequence examples:

```
chars.isGraphic( charVar );
mov( al, booleanVar );
```

```
chars.isGraphic( al );
// AL now contains the Boolean result.
```

```
mov( chars.isGraphic( ch ), booleanVar ); // Result is left in AL.
```

HLA low-level calling sequence examples:

```
mov( charVar, al );
call chars.isGraphic;
mov( al, booleanVar );
```

```
// Char to convert is in AL
```

```
call chars.isGraphic;
// Result boolean is left in AL
```

```
mov( ch, al );
call chars.isGraphic;
mov( al, booleanVar );
```

**chars.isSpace( c:byte ); @returns( "AL" );**

This routine returns true in the AL register if the parameter is a white space character. A white space character is a space, carriage return, linefeed, or tab character.

HLA high-level calling sequence examples:

```
chars.isSpace( charVar );
mov( al, booleanVar );
```

```
chars.isSpace( al );
// AL now contains the Boolean result.
```

```
mov( chars.isSpace( ch ), booleanVar ); // Result is left in AL.
```

HLA low-level calling sequence examples:

```
mov( charVar, al );
call chars.isSpace;
mov( al, booleanVar );
```

```
// Char to convert is in AL
```

```

call chars.isSpace;
// Result boolean is left in AL

mov( ch, al );
call chars.isSpace;
mov( al, booleanVar );

```

**chars.isASCII( c:byte ); @returns( "AL" );**

This routine returns true in AL if the parameter byte is an ASCII character (value in the range \$0..\$7F).

HLA high-level calling sequence examples:

```

chars.isASCII( charVar );
mov( al, booleanVar );

```

```

chars.isASCII( al );
// AL now contains the Boolean result.

```

```

mov( chars.isASCII( ch ), booleanVar ); // Result is left in AL.

```

HLA low-level calling sequence examples:

```

mov( charVar, al );
call chars.isASCII;
mov( al, booleanVar );

// Char to convert is in AL

call chars.isASCII;
// Result boolean is left in AL

mov( ch, al );
call chars.isASCII;
mov( al, booleanVar );

```

**chars.isCtrl( c:byte ); @returns( "AL" );**

This function returns true in AL if the parameter is a control character (\$0..\$1F or \$7F).

HLA high-level calling sequence examples:

```

chars.isCtrl( charVar );
mov( al, booleanVar );

```

```

chars.isCtrl( al );
// AL now contains the Boolean result.

```

```

mov( chars.isCtrl( ch ), booleanVar ); // Result is left in AL.

```

HLA low-level calling sequence examples:

```
mov( charVar, al );
call chars.isCtrl;
mov( al, booleanVar );

// Char to convert is in AL

call chars.isCtrl;
// Result boolean is left in AL

mov( ch, al );
call chars.isCtrl;
mov( al, booleanVar );
```

## 7 Console Display Control (console.hhf)

The HLA console module provides a reasonably portable way to control the console display under different operating systems. The routines in this module let you write "really-smart-terminal" console applications that behave in a similar fashion under different operating. The HLA console module routines take advantage of the Windows console API when running under Windows, they use the VT100/ANSI terminal control code sequences for other operating systems that use ANSI terminal control codes for console control. The routines in this module let you control the cursor position, erase selected portions of data from the screen, insert and delete characters and lines of text, scroll the screen, select display colors, and so on.

Note: this console module replaces the older HLA Standard Library console module that was Win32-specific. That earlier console module provided many features that are not present in the current console module because the Win32 console capabilities are quite a bit more sophisticated than is possible with an ANSI terminal emulation. Older code that took advantage of these extra features will not be able to compile properly with this new console module. The original console module is still available in the HLA distribution under the "Examples" directory; new code, however, should not use that module; if you want to take advantage of the Win32 console capabilities, you should call the Win32 API routines directly.

Note: be sure to read the chapter on "Passing Parameters to Standard Library Routines" (parmpassing.rtf) before reading this chapter.

**A Note About Thread Safety:** The args module maintains a couple of static global variables that maintain the command-line values. Currently, these values apply to all threads in a process. You should take care when changing these values in threads. The command-line is a resource that must be shared amongst all threads in an application. If you write multi-threaded applications, it is your responsibility to serialize access to the command-line functions.

### 7.1 The Console Module Module

To use the date functions in your application, you will need to include one of the following statements at the beginning of your HLA application:

```
#include( "console.hhf" )
or
#include( "stdlib.hhf" )
```

### 7.2 Cursor Positioning Functions

The functions in this category reposition the cursor on the display

```
procedure console.gotoxy( x:dword; y:dword );@pascal
procedure console.gotorc( r:dword; c:dword );@stdcall
```

*console.gotoxy* and *console.gotorc* are actually the same function. The only difference between the two calls is that they (internally) swap their parameters before making the call to the function. Note that the "rc" in *gotorc* stands for "row/column" which is equivalent to saying "gotoxy". The *console.gotorc* function was provided because people intuitively prefer to specify the row (y) value as the first argument and the column (x) value as the second argument.

These functions position the cursor at the specified (x,y)/(c,r) coordinate on the screen.

HLA high-level calling sequence examples:

```
console.gotoxy( 0, 10 );
stdout.put( "Print this on line 10, column 0" nl );
console.gotoxy( 15, 0 );
stdout.put( "Print this on line 15, column 0" nl );
```

HLA low-level calling sequence examples:

```

pushd( 0 );
pushd( 10 );
call console.gotoxy;// row = 10, column = 0

// Note that console.gotorc uses the @stdcall calling convention, so
// it's arguments are reversed from the declaration, that is, you
// push the same exact arguments you push for gotoxy (which makes
// sense, as both functions are actually the same code).

pushd( 0 );
pushd( 15 );
call console.gotorc;

```

### **procedure console.up();**

*console.up* moves the cursor up one line. Because of the variation in terminal emulations out there, the results are undefined if you execute this procedure when the cursor is on the first line of the display. Some consoles scroll the screen down one line, others ignore the request.

HLA high-level calling sequence example:

```
console.up();
```

HLA low-level calling sequence example:

```
call console.up;
```

### **procedure console.nup( n:uns32);**

*console.nup* moves the cursor up n lines. Because of the variation in terminal emulations out there, the results are undefined if this procedure attempts to move above the top line on the display. Some consoles scroll the screen down one line, others ignore the request.

HLA high-level calling sequence example:

```
console.nup( 5 );
```

HLA low-level calling sequence example:

```
pushd( 5 );
call console.nup;
```

### **procedure console.down();**

*console.down* moves the cursor down one line. Because of the variation in terminal emulations out there, the results are undefined if you execute this procedure when the cursor is on the last line of the display. Some consoles scroll the screen up one line, others ignore the request..

HLA high-level calling sequence example:

```
console.down();
```

HLA low-level calling sequence example:

```
call console.down;
```

#### **procedure console.ndown( n:uns32);**

*console.ndown* moves the cursor down one line. Because of the variation in terminal emulations out there, the results are undefined if you execute this procedure when the cursor is on the last line of the display. Some consoles scroll the screen up one line, others ignore the request.

HLA high-level calling sequence example:

```
console.ndown( 5 );
```

HLA low-level calling sequence example:

```
pushd( 5 );
call console.ndown;
```

#### **procedure console.left();**

*console.left* moves the cursor up n lines. Because of the variation in terminal emulations out there, the results are undefined if you execute this procedure and it attempts to move the cursor before the first column on the line. Some consoles move the cursor to the end of the previous line, others ignore the request.

HLA high-level calling sequence example:

```
console.left();
```

HLA low-level calling sequence example:

```
call console.left;
```

#### **procedure console.nleft( n:uns32);**

*console.nleft* moves the cursor down one line. Because of the variation in terminal emulations out there, the results are undefined if you execute this procedure when the cursor is on the last line of the display. Some consoles scroll the screen up one line, others ignore the request.

HLA high-level calling sequence example:

```
console.nleft( 5 );
```

HLA low-level calling sequence example:

```
pushd( 5 );
call console.nleft;
```

#### **procedure console.right();**

*console.right* moves the cursor to the right one character position. Because of the variation in terminal emulations out there, the results are undefined if you execute this procedure when the cursor is at the last column of the display. Some consoles move the cursor to the beginning of the next line, others ignore the request.

HLA high-level calling sequence example:

```
console.right();
```

HLA low-level calling sequence example:

```
call console.right;
```

**procedure console.nright( n:uns32);**

*console.nright* moves the cursor down one line. Because of the variation in terminal emulations out there, the results are undefined if you execute this procedure and it attempts to move the cursor beyond the last column of the display. Some consoles move the cursor to the beginning of the next line, others ignore the request.

HLA high-level calling sequence example:

```
console.nright( 5 );
```

HLA low-level calling sequence example:

```
pushd( 5 );  
call console.nright;
```

**procedure console.saveCursor();**

*console.saveCursor* saves the current cursor position in an internal variable. You can restore the cursor position via the *console.restoreCursor* call. Note that there is only one level of save available.

HLA high-level calling sequence example:

```
console.saveCursor();
```

HLA low-level calling sequence example:

```
call console.saveCursor;
```

**procedure console.restoreCursor();**

*console.restoreCursor* restores the cursor to the position previously saved by *console.saveCursor*. Note that there is only one level of "save" available.

HLA high-level calling sequence example:

```
console.restoreCursor();
```

HLA low-level calling sequence example:

```
call console.restoreCursor;
```

## 7.3 Console Clearing Functions

The functions in this category clear portions (or all) of the display.



```
procedure console.cls();  
procedure console.home();
```

*console.cls* and *console.home* are actually the same function. These procedures clear the screen and move the cursor to the home (0,0) position.

HLA high-level calling sequence examples:

```
console.cls();  
console.home();
```

HLA low-level calling sequence examples:

```
call console.cls;  
call console.home;
```

```
procedure console.clrToEOLN();
```

*console.clrToEOLN* clears the text (by writing spaces) from the current cursor position to the end of the line that the cursor is on.

HLA high-level calling sequence examples:

```
console.clrToEOLN();
```

HLA low-level calling sequence examples:

```
call console.clrToEOLN;
```

```
procedure console.clrToBOLN();
```

*console.clrToBOLN* clears the text (by writing spaces) from the current cursor position to the beginning of the line that the cursor is on.

HLA high-level calling sequence examples:

```
console.clrToBOLN();
```

HLA low-level calling sequence examples:

```
call console.clrToBOLN;
```

```
procedure console.clrLn();
```

*console.clrLn* clears the line that the cursor is on by writing spaces to that line. This does not delete the line from the screen, it only clears the characters from the line.

HLA high-level calling sequence examples:

```
console.clrLn();
```

HLA low-level calling sequence examples:

```
call console.clrLn;
```

**procedure console.clrToEOScrn();**

*console.clrToEOScrn* clears the text (by writing spaces) from the current cursor position to the end of the screen.

HLA high-level calling sequence examples:

```
console.clrToEOScrn();
```

HLA low-level calling sequence examples:

```
call console.clrToEOScrn;
```

**procedure console.clrToBOScrn();**

*console.clrToBOScrn* clears the text (by writing spaces) from the current cursor position to the beginning of the screen.

HLA high-level calling sequence examples:

```
console.clrToBOScrn();
```

HLA low-level calling sequence examples:

```
call console.clrToBOScrn;
```

## 7.4 Character Insertion/Removal Functions

The functions in this category insert and delete characters on the console display.

**procedure console.insertChar();**

*console.insertChar* inserts room for a single character by shifting the characters under the cursor and to the right of the cursor to the right one position. The vacated position is filled with a space. The last character on the line is lost.

HLA high-level calling sequence examples:

```
console.insertChar();
```

HLA low-level calling sequence examples:

```
call console.insertChar;
```

**procedure console.insertChars( n:dword );**

*console.insertChars* inserts room for *n* characters by shifting the characters under the cursor and to the right of the cursor right *n* positions. The vacated positions are filled with spaces. The last *n* characters on the line are lost.

HLA high-level calling sequence examples:

```
console.insertChars( n );
```

HLA low-level calling sequence examples:

```
pushd( n );
call console.insertChars;
```

#### **procedure console.insertLine();**

*console.insertLine* inserts a blank line before the line the cursor is on by pushing the line under the cursor, and the lines below the cursor, down one line on the screen. The new line is filled with blanks. The last line on the screen is lost.

HLA high-level calling sequence examples:

```
console.insertLine();
```

HLA low-level calling sequence examples:

```
call console.insertLine;
```

#### **procedure console.insertLines( n:dword );**

*console.insertLines* opens up n new blank lines at the current cursor position by pushing the lines at and below the cursor down n lines on the screen. The last n lines on the screen will be lost.

HLA high-level calling sequence examples:

```
console.insertLines( 5 );
```

HLA low-level calling sequence examples:

```
pushd( 5 );
call console.insertLines;
```

#### **procedure console.deleteChar();**

*console.deleteChar* deletes the character under the cursor by shifting the characters after the cursor one position to the left. The last character position at the end of the line is filled with a blank.

HLA high-level calling sequence examples:

```
console.deleteChar();
```

HLA low-level calling sequence examples:

```
call console.deleteChar;
```

**procedure console.deleteChars( n:dword );**

*console.deleteChars* deletes n characters under and to the right of the cursor by shifting the characters after the cursor n positions to the left. The n character positions at the end of the line are filled with blanks.

HLA high-level calling sequence examples:

```
console.deleteChars( n );
```

HLA low-level calling sequence examples:

```
pushd( n );
call console.deleteChars;
```

**procedure console.deleteLine();**

*console.deleteLine* deletes the line the cursor is on by shifting all the lines below the cursor position up one line. The last line on the screen is filled with blanks.

HLA high-level calling sequence examples:

```
console.deleteLine();
```

HLA low-level calling sequence examples:

```
call console.deleteLine;
```

**procedure console.deleteLines( n:dword );**

*console.deleteLines* procedure deletes n lines at and below the current cursor position. The vacated lines at the bottom of the screen are filled with blanks.

HLA high-level calling sequence examples:

```
console.deleteLines( 5 );
```

HLA low-level calling sequence examples:

```
pushd( 5 );
call console.deleteLines;
```

## 7.5 Console Scrolling

The functions in this category scroll the screen up and down.

**procedure console.scrollUp( );**

*console.scrollUp* scrolls the entire screen up one line.

HLA high-level calling sequence examples:

```
console.scrollTop();
```

HLA low-level calling sequence examples:

```
call console.scrollTop;
```

**procedure console.scrollTopDown( );**

*console.scrollTopDown* scrolls the entire screen down one line.

HLA high-level calling sequence examples:

```
console.scrollTopDown();
```

HLA low-level calling sequence examples:

```
call console.scrollTopDown;
```

## 7.6 Console Output Colors

The functions in this category control the color of the characters printed on the display.

**procedure console.setAttrs( foreground:uns32; background:uns32 );**

*console.setAttrs* sets the console internal attribute value to be used for all following character output. Use the routine to set the color of the characters you wish to print. The foreground parameter sets the color for the text characters, the background parameter sets the color of the background area of each character cell.

The console module defines the following constants that represent the corresponding colors:

```
console.black := 0;
console.red := 1;
console.green := 2;
console.yellow := 3;
console.blue := 4;
console.magenta := 5;
console.cyan := 6;
console.white := 7;
```

HLA high-level calling sequence examples:

```
console.setAttrs( console.yellow, console.blue );
```

HLA low-level calling sequence examples:

```
pushd( console.yellow );
pushd( console.blue );
call console.setAttrs;
```



## 8 Conversions (conv.hhf)

This unit contains routines that perform general conversions from one data type to another. Primarily, this unit supplies the routines that convert various data types to and from string form.

Note: be sure to read the chapter on "Passing Parameters to Standard Library Routines" (parmpassing.rtf) before reading this chapter.

Most string conversion routines take two forms: one version that writes data to an existing (preallocated) string and one that automatically allocates storage for a new string on the heap. Those functions that automatically allocate storage generally have a name that begins with "a\_" (for allocate) whereas the functions that use a preallocated string do not have such a prefix. For example, the `conv.h8ToStr` function converts an 8-bit integer to a string using hexadecimal representation and stores the result in a preallocated string object. The `conv.a_h8ToStr` function converts an 8-bit value to a (hexadecimal) string that it allocates on the heap; `conv.a_h8ToStr` returns a pointer to that string in the EAX register.

An important point to keep in mind is that string variables are pointers. Unless you call a function that allocates storage for a string (i.e., one of the "a\_..." functions), you must ensure that you've allocated sufficient storage to hold any string result the function produces. Failure to do so will produce a memory access error, null pointer reference error, or string overflow error. Remember, simply declaring a string variable does not automatically allocate storage for any string data; the declaration only allocates storage for the string pointer. You must call a function such as `str.alloc` to actually allocate the string data.

**Note about stack diagrams:** this documentation includes stack diagrams for those functions that pass parameters on the stack. To conserve space, this documentation does not include a stack diagram for any function that does not pass data on the stack (that is, only a return address appears on the stack).

**A Note About the FPU:** The Standard Library code makes occasional use of the FPU, particularly when converting between real and string formats and when computing certain mathematical functions. You should exercise caution when using MMX instructions in a program that makes use of the Standard Library. In particular, you should ensure that you are always in FPU mode (by executing an EMMS instruction) after you are finished using MMX instructions. Better yet, you should avoid the MMX instruction set altogether and use the improved SSE instruction set that accomplishes the same tasks (and doesn't disturb the FPU).

### 8.1 Buffer vs. String Conversions

The Standard Library supports two generic types of numeric-to-string conversions – output to a string variable (an "HLA string" object) and output to a memory buffer. The string conversion routines are the safest to use, but the buffer conversion routines are a bit more general.

If you're working with HLA-style string objects, then using the conversion-to-string functions make the most sense because you get to take full advantage of range checking and other facilities that are possible with the string format. Furthermore, you can use the Standard Library string manipulation functions to process such strings once the conversion is complete.

There are two drawbacks to the string conversion routines (versus the buffer conversion routines):

- You may intend to pass the converted data on to some other routine that doesn't know anything about the HLA string format, so you may need to produce the string using a different data structure.
- If you want to produce a longer string as a sequence of conversion operations, it is slightly more efficient to do the conversion to a single buffer (which may very well be an HLA string object) and fix up the string data structure afterwards.

Perhaps the most common example of a non-HLA-string data type you'll encounter is the simple zero-terminated string (the word "simple" appears here because HLA strings are zero-terminated and you can often use them whenever you need a zero-terminated string). Consider the `conv.i32ToBuf` routine that converts a 32-bit signed integer to the corresponding sequence of characters. This function stores that characters at the memory address passed in EDI and upon return EDI points at the first byte beyond the converted sequence, e.g.,

```
// Stores the characters "12345" at [edi]
```

```
conv.i32ToBuf( 12345, [edi] );
```

Upon return from this function, EDI will contain a value that is 5 greater than it was upon entry, and the five memory locations that EDI has skipped over will contain the characters "12345". Note that this string is not zero-terminated, but you can easily zero-terminate it by storing a zero byte at the location where EDI points upon return from the function:

```
// Stores the characters "12345" at [edi] and zero
// terminates the string.

conv.i32ToBuf( 12345, [edi] );
mov( 0, (type byte [edi]));
```

As a final example, suppose you want to build up an HLA style string by concatenating two converted strings together. You could do something like the following:

```
// Produces " 12345 67890" in fullStr

conv.i32ToStr( 12345, 6, ' ', leftStr );
conv.i32ToStr( 67890, 7, ' ', rightStr
str.cat( leftStr, rightStr, fullStr );
```

The only problem with this approach is that there is unnecessary string processing (e.g., data copying) taking place. If efficiency is paramount, and you don't need the intermediate conversions (leftStr and rightStr), then you can do this sequence a little bit faster by generating the two strings in place as follows:

```
mov( fullStr, edi );           // Point EDI at start of string data
mov( edi, ebx );              // Save to compute length
conv.i32ToBuf( 12345, 6, ' ' ); // Store " 12345" at fullStr
conv.i32ToBuf( 67890, 7, ' ' ); // Store " 67890" at fullStr+6
mov( 0, (type byte [edi]));    // HLA strings must be zero terminated
sub( fullStr, edi );           // Compute string length
mov( edi, (type str.strRec [ebx]).length ); // Save new length.
```

As the number of objects you append to the string increases, this scheme becomes even more efficient than using the str.cat approach. The code above, of course, assumes that you've already allocated a sufficient amount of string storage for the leftStr, rightStr, and fullStr string variables.

## 8.2 Conversion Format Control

The following functions control the numeric conversion process.

### 8.2.1 Underscore Control

When converting numeric data types to strings, the standard library offers the option of inserting underscores at appropriate places in the numbers (i.e., where you would normally expect a digit separator to



appear, such as a comma [U.S.] or period [Europe]). The `conv.setUnderscores` and `conv.getUnderscores` functions control the operation of this feature.

The standard library conversion functions will inject underscores into hexadecimal, unsigned integers, and signed integers if the feature is enabled. For hexadecimal output the standard library conversion routines will emit an underscore between every fourth and fifth digit, starting with the L.O. digit (e.g., 1234\_5678). For decimal integers (signed or unsigned), the conversion routines emit an underscore between each third and fourth digit starting with the L.O. digit (e.g., 123\_456\_789).

Note that the conversion routines do *not* emit underscores into conversions of floating-point/real values.

**Thread Issues:** Because the standard library maintains the internal underscore flag as a static object there will be some problems if you attempt to read and set the underscore flag in multiple threads running in the same address space. In particular, if you read the underscores flag and save it, set it to a different value, do some conversions, and then restore the underscores flag its original value, it is quite possible that another thread could do some conversions between those two points and produce incorrect output. Indeed, it would even be possible for half the number to contain underscores and the other half not contain underscores, depending on where the system interrupts the second thread. The current library code does not address this issue because the cost is very high to solve a problem that almost never occurs (most assembly applications are single-threaded). However, if you are writing a multi-threaded application, you should note that constantly changing the underscores flag is not a good idea – you should try to set the flag once, at the beginning of your program, and leave it alone throughout the program's execution. If you must change the underscore flag setting on a regular basis within a multi-threaded application, you should put appropriate locks around all calls to conversion routines (and those routines, such as the I/O routines, that call the conversion code) to protect the settings.

Current plans are to make the Standard Library thread-safe when the threads module is added to the library.

**`conv.setUnderscores( onOff: boolean );`**

The `conv.setUnderscores` function lets you enable or disable the emission of underscores in numeric values. Passing true enables underscore emission, passing false disables it.

For efficiency reasons, the standard library routines always pass all parameters as a multiple of four bytes. The *onOff* Boolean parameter consumes the L.O. byte of the double word passed on the stack. The `conv.setUnderscores` routine ignores the H.O. three bytes of the value passed for this parameter, though by convention (to make debugging a little easier) you should try to pass zeros in the H.O. three bytes if it is not inconvenient to do so.

When passing a Boolean constant, you should simply push the dword containing the value true (1) or false (0), e.g.,

```
pushd( true );
call conv.setUnderscores;
.
.
.
pushd( false );
call conv.setUnderscores;
```

When passing the Boolean value in one of the 8-bit registers AL, BL, CL or DL, you should simply push the 32-bit register that holds the 8-bit register, e.g.,

```
push( eax ); // Pushes AL onto the stack
call conv.setUnderscores;
push( ebx ); // Pushes BL onto the stack
call conv.setUnderscores;
```

Note that this trick does not apply to the AH, BH, CH, or DH registers. The best code to use when you need to push these registers is to drop the stack down by four bytes and then move the desired register into the memory location you've just created on the stack, e.g.,

```
sub( 4, esp );
mov( AH, [esp] ); // Pushes AH onto the stack
call conv.setUnderscores;
.
```

```

.
.
sub( 4, esp );
mov( BH, [esp] ); // Pushes BH onto the stack
call conv.setUnderscores;

```

Here's another way you can accomplish this (a little slower, but leaves zeros in the H.O. three bytes):

```

pushd( 0 );
mov( CH, [esp] ); // Pushes CH onto the stack
call conv.setUnderscores;

```

When passing a Boolean variable, you should try to push the Boolean value and the following three bytes, using code like the following (HLA syntax):

```

pushd( (type dword boolVar) );
call conv.setUnderscores;

```

There is one drawback to the approach above. In three very rare cases the code above could cause a segmentation fault. If the Boolean variable is located on the last three bytes of a page in memory (4,096 bytes) and the next memory page is not readable, the system will generate a fault if you attempt to push all four bytes. In such a case, the next best solution, if a register is available, is to move the Boolean value into AL, BL, CL, or DL and push the corresponding 32-bit register. If no registers are available, then you can write code like the following:

```

push( eax );
push( eax );
mov( boolVar, al );
mov( al, [esp+4] );
pop( eax );
call conv.setUnderscores;

```

This code is ugly and slightly inefficient, but it will always work (assuming, of course, you don't get a stack overflow).

The HLA compiler will generate code similar to this last example if you pass a boolean variable as the actual parameter to `conv.setUnderscores`:

```

conv.setUnderscores( boolVar );

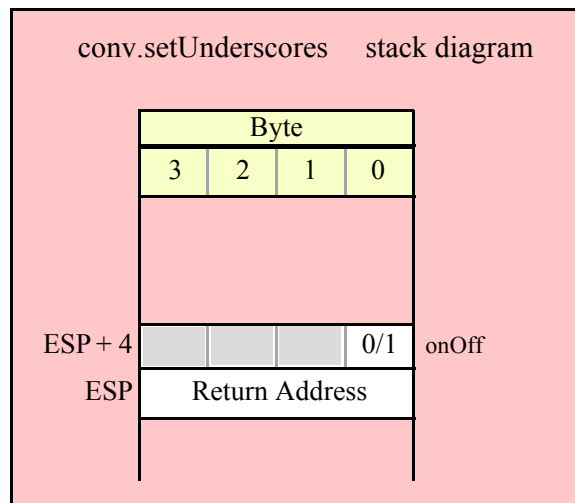
```

Therefore, if efficiency is a concern to you, you should try to load the Boolean variable (`boolVar` in this example) into AL, BL, CL, or DL prior to calling `conv.setUnderscores`, e.g.,

```

mov( boolVar, al );
conv.setUnderscores( al );

```



```
conv.getUnderscores; @returns( "eax" );
```

You can test the current state of the underscore conversion by calling `conv.getUnderscores`. This function call returns the boolean result in EAX (true means underscores will be output); AL will contain the actual Boolean value and the H.O. three bytes of EAX will all contain zero. This routine does not have any parameters.

The following example demonstrates how to preserve the value of the internal underscores flag across some section of code:

```
conv.getUnderscores();
mov( al, saveUnderscores );
conv.setUnderscores( true );
.
.
.
mov( saveUnderscores, al );
conv.setUnderscores( al );
```

**Note:** Do not try to access the internal underscores flag directly in your program. Always use the `conv.setUnderscores` and `conv.getUnderscores` accessor functions. In a future version of the Standard Library, the internal representation of this flag *will* change and any code that accesses it directly will break at that point. However, if you call `conv.setUnderscores` and `conv.getUnderscores`, you're guaranteed that the internal implementation will be hidden from you and your code will not fail when the internal representation changes.

## 8.2.2 Delimiter Control

During the conversion from string to a numeric form, the conversion routines will skip over zero or more delimiter characters and then process all numeric digits (including hexadecimal digits, if doing a hexadecimal conversion) up to the end of string or the first delimiter character it finds. If a conversion function encounters a value that is not a valid digit or delimiter character, it will raise a conversion exception or an illegal character exception. By default, the delimiter characters are members of the following set:

```
Delimiters: cset :=
{
    #0,    // End of string
    #9,    // Tab
```

```

    #10,    // Line feed
    #13,    // Carriage return
    ' ',    // Space
    ',',    // Comma
    ';',    // Semicolon
    ':',    // Colon
};

```

While this default delimiters character set is probably appropriate for most applications, some programmers may want to add or remove characters from this set based on their application requirements. The standard library provides two routines that provide access to this internal character set object: `conv.getDelimiters` and `conv.setDelimiters`. You should always use these routines to access this character set object rather than accessing it directly (as an external object).

**Thread Issues:** Because the standard library maintains the internal delimiters character set as a static object there will be some problems if you attempt to read and set the delimiters in multiple threads running in the same address space. In particular, if you read the delimiters character set and save it, set it to a different value, do some conversions, and then restore the delimiters to the original value, it is quite possible that another thread could do some conversions between those two points and produce incorrect. The current library code does not address this issue because the cost is very high to solve a problem that almost never occurs (most assembly applications are single-threaded). However, if you are writing a multi-threaded application, you should note that constantly changing the delimiters character set is not a good idea – you should try to set the delimiters once, at the beginning of your program, and leave them alone throughout the program's execution. If you must change the delimiters character set on a regular basis within a multi-threaded application, you should put appropriate locks around all calls to conversion routines (and those routines, such as the I/O routines, that call the conversion code) to protect the settings.

Current plans are to make the delimiters character set object thread-safe when the processes module is added to the library.

**Note:** Do not try to access the internal delimiters character set directly in your program. Always use the `conv.setDelimiters` and `conv.getDelimiters` accessor functions. In a future version of the Standard Library, the internal representation of this character set *will* change and any code that accesses it directly will break at that point. However, if you call `conv.setDelimiters` and `conv.getDelimiters`, you're guaranteed that the internal implementation will be hidden from you and your code will not fail when the internal representation changes.

```
conv.getDelimiters( var Delims: cset );
```

The `conv.getDelimiters` routine returns the current delimiters character set in the variable you pass by reference as the parameter. The *Delims* parameter is passed by reference (that is, you pass the address of the actual `cset` variable to receive the result). The following are examples of typical HLA high-level invocations of this routine looks like this:

```

conv.getDelimiters( saveDelims );

// EDI points at the delimiter cset:

conv.getDelimiters( [edi] );

// ptrToDelims is a dword/pointer variable that contains
// the address of a cset object:

conv.getDelimiters( val ptrToDelims );

```

To call `conv.getDelimiters` using a low-level assembly syntax, you must push the address of a `cset` variable object onto the stack and then call the `conv.getDelimiters` function:

```

// saveDelims_s is a variable declared in the static/storage section:

pushd( &saveDelims_s );
call conv.getDelimiters;

```

```
// saveDelims_v is a variable declared in the var section or
// is a parameter:

lea( eax, saveDelims_v );
push( eax );
call conv.getDelimiters;

// Alternative call passing saveDelims_v if no 32-bit registers
// are available (this code assumes that EBP points at the current
// activation record/stack frame that contains saveDelims_v):

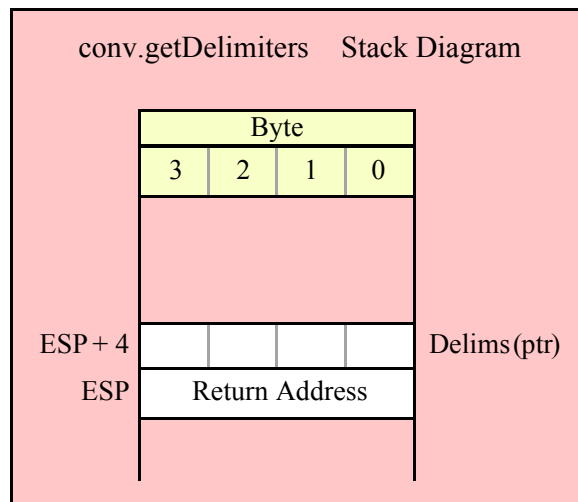
push( ebp );
add( @offset( saveDelims_v ), (type dword [esp] ));
call conv.getDelimiters;

// Low-level call assuming a 32-bit register (esi in this case)
// contains the address of the cset:

push( esi );
call conv.getDelimiters;

// Low-level call assuming a dword or pointer variable contains the
// address of the cset that will receive the delimiter character set:

push( ptrToDelims );
call conv.getDelimiters;
```



```
conv.setDelimiters( Delims: cset )
```

The `conv.setDelimiters` function lets you change the value of the internal delimiter character set. It requires a 16-byte character set parameter (passed by value) and will copy the value of this parameter to the internal character set variable. Note that this routine makes a copy of the actual parameter you pass it. If you pass an character set variable as the actual parameter, future changes to that character set variable (without a corresponding call to `conv.setDelimiters`) will have no effect on the internal delimiters character set that the standard library routines use. The following examples are typical HLL style calls to this function:

```
conv.setDelimiters( { ' ', '\', '}' );// Pass in a literal constant
conv.setDelimiters( csetVar );// Pass in a cset variable's value
conv.setDelimiters( [edx] );// EDX points at a cset variable
```

To call `conv.setDelimiters` using a low-level calling sequence, you'd first push the 16 bytes associated with the character set object (H.O. dword first down to the L.O. dword) and then call the `conv.setDelimiters` function. Here are some examples:

```
// Push the literal cset constant {' ', ','} onto the stack:

pushd( 0 ); // Must manually convert cset to a sequence of
pushd( $1001 ); // four dwords (ugh!). Note: ORD( ' ' ) = $20
pushd( 0 ); // and ORD( ',' ) = $2C so bit positions $20 and
pushd( 0 ); // $2c must contain '1's, zeros everywhere else.
call conv.setDelimiters;


// Push the cset variable "saveDelims" onto the stack and
// call conv.setDelimiters:

push( (type dword saveDelims[12]));
push( (type dword saveDelims[8]) );
push( (type dword saveDelims[4]) );
push( (type dword saveDelims[0]) );
call conv.setDelimiters.


// If manually converting a literal cset constant to the equivalent
// numeric values isn't your thing, you can also do the following
// (though this is slightly less efficient):

readonly
    spaceAndComma :cset := { ' ', ',' };
endreadonly

push( (type dword spaceAndComma [12]));
push( (type dword spaceAndComma [8]) );
push( (type dword spaceAndComma [4]) );
push( (type dword spaceAndComma [0]) );
call conv.setDelimiters.
```

If you insist on using low-level calling sequences to call the `conv.setDelimiters` routine, you might want to consider writing a macro that will automatically push a literal `cset` constant for you. Here is a set of HLA macros that will do this task:

```

program t;

// dword_n extracts the nth dword (0, 1, 2, 3) from a
// 16-byte object such as a character set. cst must be
// a cset constant value (or an lword), n must be an
// integer constant in the range 0..3.

#define dword_n( cst, n );

    (
        (@byte( @lword(cst), n*4+3) << 24)
        | (@byte( @lword(cst), n*4+2) << 16)
        | (@byte( @lword(cst), n*4+1) << 8)
        | (@byte( @lword(cst), n*4+0) << 0)
    )
#definemacro

// pushcset pushes the cset constant passed as an argument
// onto the CPU's stack. H.O. dword is pushed first, L.O.
// dword is pushed last.

```

```

macro pushcset( cst );

    // Push the four dwords that make up a cset constant:

    pushd( dword_n( cst, 3 ) );
    pushd( dword_n( cst, 2 ) );
    pushd( dword_n( cst, 1 ) );
    pushd( dword_n( cst, 0 ) );

endmacro

begin t;

    // Example of pushcset invocation:

    pushcset( { ' ', ' ', ' ', ' ' } );

end t;

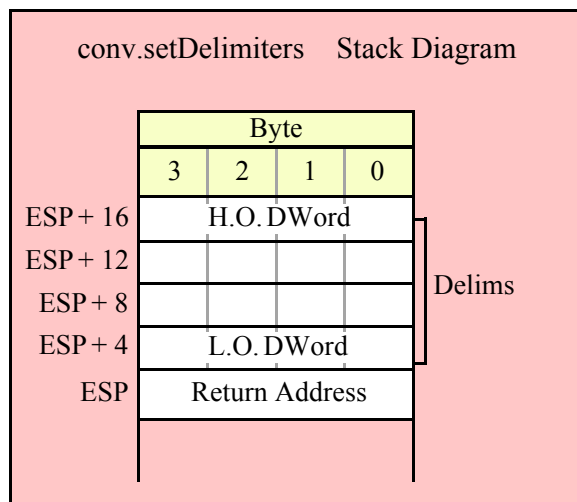
```

Here is an example using `conv.getDelimiters` and `conv.setDelimiters` that demonstrates how to temporarily change the delimiters character set and then restore its value:

```

var
    saveDelims:cset;
.
.
.
conv.getDelimiters( saveDelims );
conv.setDelimiters( { '!', '@' } )1
.
.
.
conv.setDelimiters( saveDelims );

```



## 8.3 Hexadecimal Conversions

The standard library hexadecimal routines convert numeric values of varying sizes (8, 16, 32, 64, 80, and 128 bits) into a string of characters holding the hexadecimal representation of those values. The hexadecimal output routines can be broken down into the following categories:

Output type (string or sequence of characters to a buffer)

Fill type (no fill; fill with zeros to

a standardized length, based on data type; fill with a caller-specified character to a caller-specified length).

### 8.3.1 Internal Routines

The conversions module in the standard library contains several routines that are intended for internal use only. Generally, you should not call these routines directly from your application programs. These routines all have names that begin with an underscore. The internal hexadecimal conversion routines include:

`_hexTobuf64Size`, `_hexTobuf80`, `_hexTobuf80Size`, `_hexTobuf128`, `_hexTobuf128Size`, `_hexTobuf32`, `_hexTobuf32Size`, and `_hexTobuf64`.

### 8.3.2 Hexadecimal Numeric Size Functions

The hexadecimal conversion size functions return the number of digit print positions required by the conversion of a numeric value to a hexadecimal string. There are two sets of six routines that compute the output size: one set computes the fixed-size width and the other set computes a varying-sized width.

#### 8.3.2.1 Fixed Size Hexadecimal Size Functions

It is common practice to display hexadecimal values using exactly one digit for each nibble of the corresponding data type, including leading zeros, as necessary. The common fixed sizes are `byte=2`, `word=4`, `dword=8`, `qword=16`, `tbyte=20`, and `lword=32`. With underscore output enabled (see `conv.setUnderscores`) these values are `byte=2`, `word=4`, `dword=9`, `qword=19`, `tb=24`, and `lword=39`. Because these numbers are fixed (at least, for a given underscores flag setting) there are only three reasons for calling these functions:

- You don't know the underscores flag setting when executing a particular section of code (which can affect the output size of `dword`, `qword`, `tbyte`, and `lword` objects), or,
- You're generating a call to these functions via some macro that is given a function name like "putb" or "puth8" and you're manually constructing the size function to call via the assembler's compile-time language. or,
- You're writing generic code and you want to make it easy to modify the code in the future.

```
procedure conv.bSize( b:byte in al ); @returns( "eax" );
```

This function always returns 2 because the fixed output size of a byte is two hexadecimal digits (two 4-bit nibbles).

HLA high-level calling sequence examples:

```
conv.bSize( byteVariable );
conv.bSize( <byte register> ); // al, ah, bl, bh, cl, ch, dl, dh
conv.bSize( <constant> );      // Must fit into eight bits
```

Because `conv.bSize` passes its input parameter in the AL register, any form of the high-level calling sequence except "`conv.bSize( al );`" will automatically generate an instruction of the form "`mov(<operand>,al);`". Therefore, if at all possible, you should try to have the value whose size you wish to compute already sitting in the AL register and pass AL as the parameter to `conv.bSize`.

HLA low-level calling sequence examples:

```
mov( byteVariable, al );
call conv.bSize;

mov( <byte register>, al ); // ah, bl, bh, cl, ch, dl, dh
call conv.bSize;

mov( <constant>, al );
call conv.bSize;
```



```
procedure conv.wSize( w:word in ax ); @returns( "eax" );
```

This function always returns 4 because the "natural" size of a word is four hexadecimal digits (four 4-bit nibbles).

HLA high-level calling sequence examples:

```
conv.wSize( wordVariable );
conv.wSize( <word register> ); // ax, bx, cx, dx, bp, sp, si, di
conv.wSize( <constant> );      // Must fit into 16 bits
```

Because conv.wSize passes its input parameter in the AX register, any form of the high-level calling sequence except "conv.wSize( ax );" will automatically generate an instruction of the form "mov(<operand>,ax);". Therefore, if at all possible, you should try to have the value whose size you wish to compute already sitting in the AX register and pass AX as the parameter to conv.wSize.

HLA low-level calling sequence examples:

```
mov( wordVariable, ax );
call conv.wSize;

mov( <word register>, ax ); // bx, cx, dx, bp, sp, si, di
call conv.wSize;

mov( <constant>, ax );
call conv.wSize;
```

```
procedure conv.dSize( d:dword in eax ); @returns( "eax" );
```

This function returns 8 if the internal underscores flag is false, 9 if it is true, because the "natural" size of a double word is eight hexadecimal digits (eight 4-bit nibbles).

HLA high-level calling sequence examples:

```
conv.dSize( dwordVariable );
conv.dSize( <dword register> ); // eax, ebx, ecx, edx,
                                //ebp, esp, esi, edi
conv.dSize( <constant> );      // Must fit into 32 bits
```

Because conv.dSize passes its input parameter in the EAX register, any form of the high-level calling sequence except "conv.dSize( eax );" will automatically generate an instruction of the form "mov(<operand>,eax);". Therefore, if at all possible, you should try to have the value whose size you wish to compute already sitting in the EAX register and pass EAX as the parameter to conv.dSize.

HLA low-level calling sequence examples:

```
mov( dwordVariable, eax );
call conv.dSize;

mov( <dword register>, eax ); // ebx, ecx, edx, ebp, esp, esi, edi
call conv.dSize;

mov( <constant>, eax );
call conv.dSize;
```

```
procedure conv.qSize( q:qword ); @returns( "eax" );
```

This function returns 16 if the internal underscores flag is false, 19 if it is true, because the "natural" size of a quad word is 16 hexadecimal digits (16 4-bit nibbles) and there are four groups of four digits with underscores between them (if the underscores flag contains true).

HLA high-level calling sequence examples:

```
conv.qSize( qwordVariable );
conv.qSize( <constant> );           // Must fit into 64 bits
```

HLA low-level calling sequence examples:

```
// Passing a qword variable

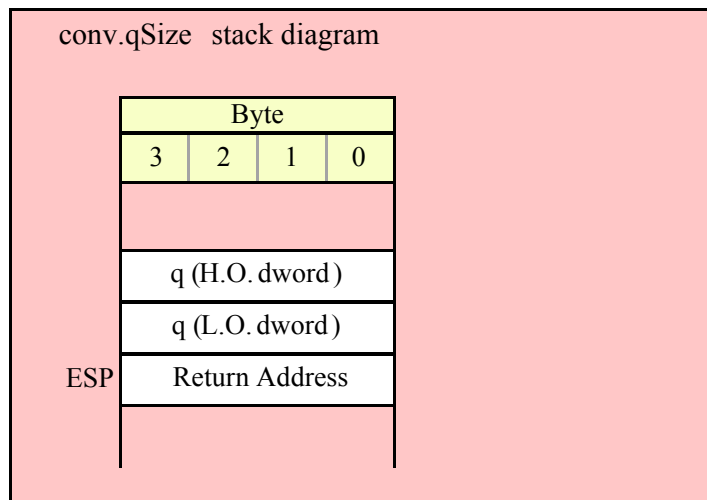
push( (type dword qwordVar[4]) ); // Push H.O. dword first
push( (type dword qwordVar[0]) ); // Push L.O. dword second
call conv.qSize;

// Example where 64-bit value is held in EDX:EAX

push( edx ); // Push H.O. dword first
push( eax ); // Push L.O. dword second
call conv.qSize;

// Passing a qword constant:

pushd( <qword constant> >> 32 ); // Push H.O. dword
pushd( <qword constant> & $FFFF_FFFF ); // Push L.O. dword
call conv.qSize;
```



```
procedure conv.tbSize( tb:tbyte ); @returns( "eax" );
```

This function returns 20 if the internal underscores flag is false, 24 if it is true, because the "natural" size of a ten-byte word is 20 hexadecimal digits (20 4-bit nibbles) and there are five groups of four digits with underscores between them (if the underscores flag contains true).

HLA high-level calling sequence examples:

```
conv.tbSize( tbyteVariable );
conv.tbSize( <constant> );           // Must fit into 80 bits
```

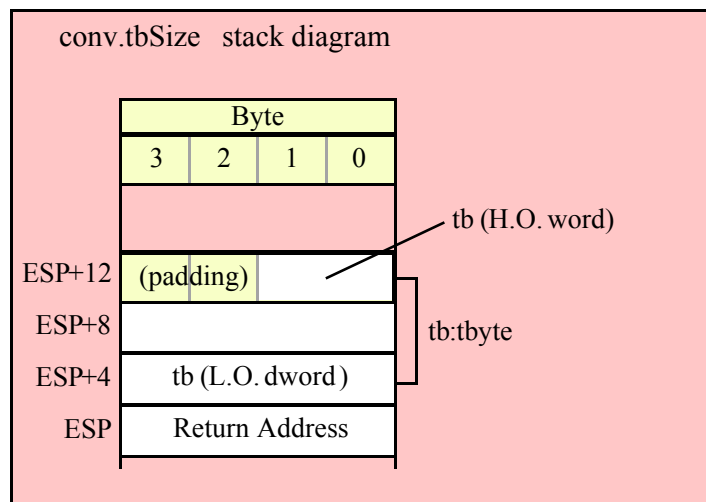
HLA low-level calling sequence examples:

```
// Passing a tbyte variable
```

```
pushw( 0 );                               // Must pad parameter to 12 bytes
push( (type word tbyteVar[8]) );          // Push H.O. word
push( (type dword tbyteVar[4]) );         // Push middle dword second
push( (type dword tbyteVar[0]) );         // Push L.O. dword third
call conv.tbSize;
```

```
// Passing a tbyte constant:
```

```
pushw( 0 ); // Must pad to 12 bytes.
pushw( <tbyte constant> >> 64 );          // Push H.O. word
pushd( (<tbyte constant> >> 32) & $FFFF_FFFF ); // Push middle dword
pushd( <tbyte constant> & $FFFF_FFFF );     // Push L.O. dword
call conv.tbSize;
```



```
procedure conv.lSize( l:ldword ); @returns( "eax" );
```

This function returns 32 if the internal underscores flag is false, 39 if it is true, because the "natural" size of an ldword is 32 hexadecimal digits (32 4-bit nibbles) and there are eight groups of four digits with underscores between them (if the underscores flag contains true).

HLA high-level calling sequence examples:

```
conv.lSize( ldwordVariable );
conv.lSize( <constant> );           // Must fit into 128 bits
```

HLA low-level calling sequence examples:

```
// Passing an ldword variable
```

```
push( (type dword ldwordVar[12]));       // Push H.O. dword first
```

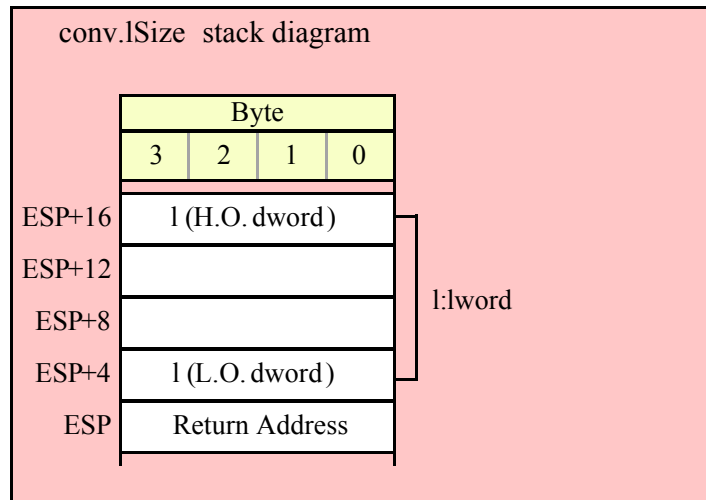
```

push( (type dword tbyteVar[8]) );
push( (type dword tbyteVar[4]) );
push( (type dword tbyteVar[0]) );    // Push L.O. dword last
call conv.lSize;

// Passing a lword constant:

pushd( <lword constant> >> 96 );    // Push H.O. dword first
pushw( (<lword constant> >> 64) & $FFFF_FFFF );
pushd( (<lword constant> >> 32) & $FFFF_FFFF );
pushd( <lword constant> & $FFFF_FFFF ); // Push L.O. dword last
call conv.lSize;

```



### 8.3.2.2 Standard Hexadecimal Size Functions

The h8Size, h16Size, h32Size, h64Size, h80Size, and h128Size routines compute the minimum number of output hexadecimal digits (with no leading zeros). These functions return a count that includes space for underscores if the internal underscores flag contains true (see conv.setUnderscores for details).

```

procedure conv.h8Size( b:byte in al ); @returns( "eax" );

```

This function returns the number of print positions required by the conversion of the value in AL to a string of hexadecimal digits. The return value is always 1 or 2 (as a single byte never consumes more than two hexadecimal digits). Note that the internal underscores flag setting does not affect the return result because the conversion routines never inject an underscore into a hexadecimal conversion unless there are at least five output digits.

HLA high-level calling sequence examples:

```

conv.h8Size( byteVariable );
conv.h8Size( <byte register> ); // al, ah, bl, bh, cl, ch, dl, dh
conv.h8Size( <constant> );     // Must fit into eight bits

```

Because conv.h8Size passes its input parameter in the AL register, any form of the high-level calling sequence except "conv.h8Size( al );" will automatically generate an instruction of the form "mov(<operand>,al);". Therefore, if at all possible, you should try to have the value whose size you wish to compute already sitting in the AL register and pass AL as the parameter to conv.h8Size.

HLA low-level calling sequence examples:

```

mov( byteVariable, al );
call conv.h8Size;

mov( <byte register>, al ); // ah, bl, bh, cl, ch, dl, dh
call conv.h8Size;

mov( <constant>, al );
call conv.h8Size;

```

**procedure conv.h16Size( w:word in ax ); @returns( "eax" );**

This function returns the number of print positions required by the conversion of the value in AX to a string of hexadecimal digits. The return value is always 1, 2, 3, or 4 (as a single word never requires more than four hexadecimal digits). Note that the internal underscores flag setting does not affect the return result because the conversion routines never inject an underscore into a hexadecimal conversion unless there are at least five output digits.

HLA high-level calling sequence examples:

```

conv.h16Size( wordVariable );
conv.h16Size( <word register> ); // ax, bx, cx, dx, bp, sp, si, di
conv.h16Size( <constant> );      // Must fit into 16 bits

```

Because conv.h16Size passes its input parameter in the AX register, any form of the high-level calling sequence except "conv.h16Size( ax );" will automatically generate an instruction of the form "mov(<operand>,ax);". Therefore, if at all possible, you should try to have the value whose size you wish to compute already sitting in the AX register and pass AX as the parameter to conv.h16Size.

HLA low-level calling sequence examples:

```

mov( wordVariable, ax );
call conv.h16Size;

mov( <word register>, ax ); // bx, cx, dx, bp, sp, si, di
call conv.h16Size;

mov( <constant>, ax );
call conv.h16Size;

```

**procedure conv.h32Size( s:dword in eax ); @returns( "eax" );**

This function returns the number of print positions required by the conversion of the value in EAX to a string of hexadecimal digits. The return value is always in the range 1-8 if the internal underscores flag is false, it is in the range 1-9 if the internal underscores flag is true (see the discussion of conv.setUnderscores for details).

HLA high-level calling sequence examples:

```

conv.h32Size( dwordVariable );
conv.h32Size( <dword register> ); // eax, ebx, ecx, edx,
                                //ebp, esp, esi, edi
conv.h32Size( <constant> );      // Must fit into 32 bits

```

Because conv.h32Size passes its input parameter in the EAX register, any form of the high-level calling sequence except "conv.h32Size( eax );" will automatically generate an instruction of the form "mov(<operand>,eax);". Therefore, if at all possible, you should try to have the value whose size you wish to compute already sitting in the EAX register and pass EAX as the parameter to conv.h32Size.

HLA low-level calling sequence examples:

```

mov( dwordVariable, eax );
call conv.h32Size;

mov( <dword register>, eax ); // ebx, ecx, edx, ebp, esp, esi, edi
call conv.h32Size;

mov( <constant>, eax );
call conv.h32Size;

```

**procedure conv.h64Size( q:qword ); @returns( "eax" );**

This function returns the number of print positions required by the conversion of the value passed in q to a string of hexadecimal digits. The return value is always in the range 1-16 if the internal underscores flag is false, it is in the range 1-19 if the internal underscores flag is true (see the discussion of conv.setUnderscores for details).

HLA high-level calling sequence examples:

```

conv.h64Size( qwordVariable );
conv.h64Size( <constant> );           // Must fit into 64 bits

```

HLA low-level calling sequence examples:

// Passing a qword variable

```

push( (type dword qwordVar[4]) ); // Push H.O. dword first
push( (type dword qwordVar[0]) ); // Push L.O. dword second
call conv.h64Size;

```

// Example where 64-bit value is held in EDX:EAX

```

push( edx ); // Push H.O. dword first
push( eax ); // Push L.O. dword second
call conv.h64Size;

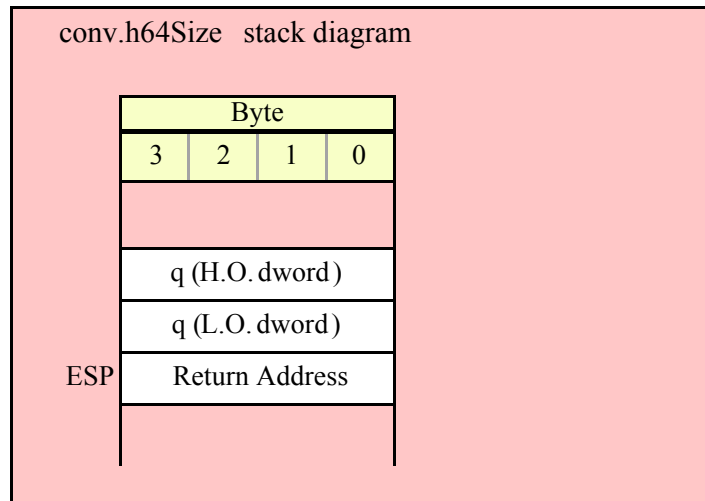
```

// Passing a qword constant:

```

pushd( <qword constant> >> 32 );           // Push H.O. dword
pushd( <qword constant> & $FFFF_FFFF );    // Push L.O. dword
call conv.h64Size;

```



```
procedure conv.h80Size( tb:tbyte ); @returns( "eax" );
```

This function returns the number of print positions required by the conversion of the value passed in tb to a string of hexadecimal digits. The return value is always in the range 1-20 if the internal underscores flag is false, it is in the range 1-24 if the internal underscores flag is true (see the discussion of conv.setUnderscores for details).

HLA high-level calling sequence examples:

```
conv.h80Size( tbyteVariable );
conv.h80Size( <constant> );           // Must fit into 80 bits
```

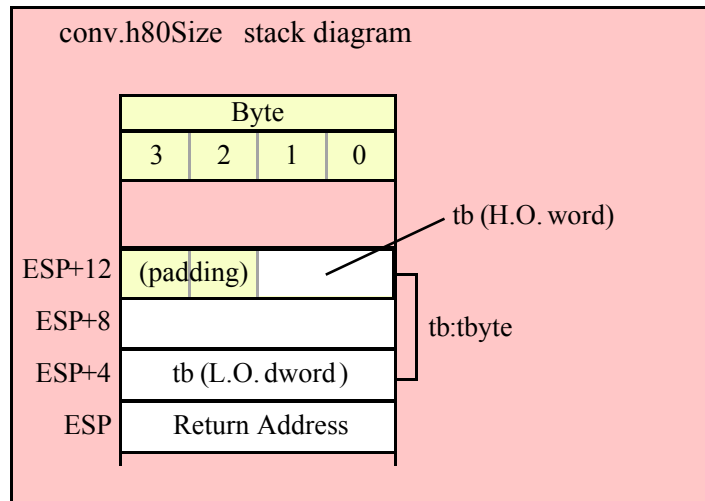
HLA low-level calling sequence examples:

```
// Passing a tbyte variable
```

```
pushw( 0 );                               // Must pad parameter to 12 bytes
push( (type word tbyteVar[8]) );          // Push H.O. word
push( (type dword tbyteVar[4]) );         // Push middle dword second
push( (type dword tbyteVar[0]) );         // Push L.O. dword third
call conv.h80Size;
```

```
// Passing a tbyte constant:
```

```
pushw( 0 ); // Must pad to 12 bytes.
pushw( <tbyte constant> >> 64 );          // Push H.O. word
pushd( (<tbyte constant> >> 32) & $FFFF_FFFF ); // Push middle dword
pushd( <tbyte constant> & $FFFF_FFFF );     // Push L.O. dword
call conv.h80Size;
```



```
procedure conv.h128Size( l:lword ); @returns( "eax" );
```

This function returns the number of print positions required by the conversion of the value passed in *l* to a string of hexadecimal digits. The return value is always in the range 1-32 if the internal underscores flag is false, it is in the range 1-39 if the internal underscores flag is true (see the discussion of `conv.setUnderscores` for details).

HLA high-level calling sequence examples:

```
conv.h128Size( lwordVariable );
conv.h128Size( <constant> );           // Must fit into 128 bits
```

HLA low-level calling sequence examples:

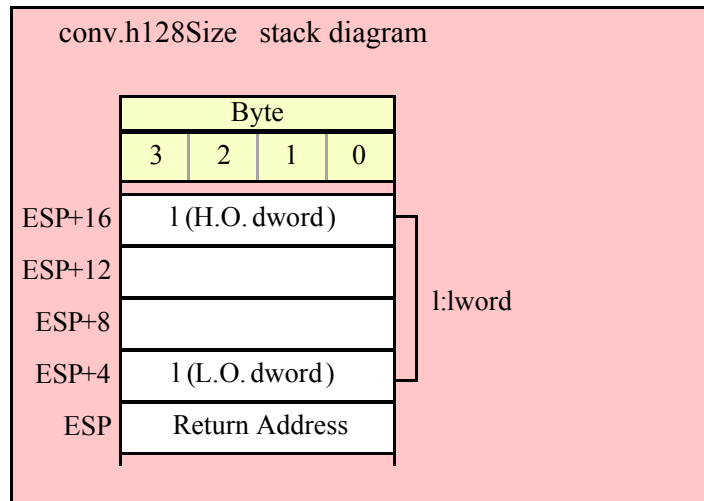
```
// Passing an lword variable
```

```
push( (type dword lwordVar[12])); // Push H.O. dword first
push( (type dword tbyteVar[8]) );
push( (type dword tbyteVar[4]) );
push( (type dword tbyteVar[0]) ); // Push L.O. dword last
call conv.h128Size;
```

```
// Passing a lword constant:
```

```
pushd( <lword constant> >> 96 ); // Push H.O. dword first
pushw( (<lword constant> >> 64) & $FFFF_FFFF );
pushd( (<lword constant> >> 32) & $FFFF_FFFF );
pushd( <lword constant> & $FFFF_FFFF ); // Push L.O. dword last
call conv.h128Size;
```





### 8.3.3 Hexadecimal Numeric to Buffer Conversions

The hexadecimal numeric to buffer conversion routines translate a numeric value to a sequence of hexadecimal characters and store those characters into memory starting at the address pointed at by the EDI register. After the conversion, the EDI register points at the first byte in memory beyond the sequence of characters these routines produce. With successive calls to these routines (or any routine that emits characters to the buffer at which EDI points) you can build up larger strings.

If the internal underscores flag is true, these routines will emit an underscore between each group of four hexadecimal digits.

#### 8.3.3.1 Fixed Length Hexadecimal Numeric to Buffer Conversions

The fixed length hexadecimal to buffer conversion routines translate an input numeric value to a fixed-length string (depending on the data type size and the setting of the internal underscores flag). These functions emit the characters of the string (including leading zeros, as necessary) to sequential locations starting at the address held in EDI.

```
procedure conv.bToBuf( b:byte in al; var buffer:var in edi );
```

Converts the numeric value in AL to a sequence of exactly two hexadecimal digits (including a leading zero if the value is in the range \$0-\$f) and stores these two characters at the location pointed at by EDI. This function returns EDI pointing at the first byte after the sequence. Because the conversion is always less than five digits, the internal underscores flag does not affect the output this function produces.

HLA high-level calling sequence examples:

```
// The following will load "byteVariable" into AL and
// the address of "charArrayVariable" into EDI and then
// call conv.bToBuf:
```

```
conv.bToBuf( byteVariable, charArrayVariable );
```

```
// The following call will copy BH into AL and
// EDX into EDI prior to calling conv.bToBuf:
```

```
conv.bToBuf( bh, [edx] );
```

```
// The following just calls conv.bToBuf as AL and EDI
// already hold the parameter values:
```

```

conv.bToBuf( al, [edi] );

// The following loads the constant in AL and calls
// conv.bToBuf:

conv.bToBuf( <constant>, [edi] ); // <constant> must fit in 8 bits

```

When using the HLA high-level calling form, always keep in mind that these statements load the AL and EDI registers with their respective parameter values. In particular, you should not specify [EAX] as the buffer address because the code that HLA generates can overwrite the L.O. byte of EAX (i.e., AL) before it copies the address to the EDI register. It goes without saying that this function will overwrite the values of EAX and EDI if the original parameters are not AL and [EDI].

HLA low-level calling sequence examples:

```

// Passing a byte variable and a buffer variable:

mov( byteVariable, al );
lea( edi, charArrayVariable );
call conv.bToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

mov( byteVariable, al );
mov( &charArrayVariable, edi );
call conv.bToBuf;

// Passing a pair of registers (that are not
// AL and EDI):

mov( bh, al );
mov( edx, edi );
call conv.bToBuf;

// Passing a constant:

mov( <constant>, al );
call conv.bToBuf; // Assume EDI already contains buffer address.

```

**procedure conv.wToBuf( w:word in ax; var buffer:var in edi );**

Converts the numeric value in AX to a sequence of exactly four hexadecimal digits (including leading zeros, as necessary) and stores these four characters at the location pointed at by EDI. This function returns EDI pointing at the first byte after the sequence. Because the conversion is always less than five digits, the internal underscores flag does not affect the output this function produces.

HLA high-level calling sequence examples:

```

// The following will load "wordVariable" into AX and
// the address of "charArrayVariable" into EDI and then
// call conv.wToBuf:

conv.wToBuf( wordVariable, charArrayVariable );

// The following call will copy BX into AX and
// EDX into EDI prior to calling conv.wToBuf:

conv.wToBuf( bx, [edx] );

```

```
// The following just calls conv.wToBuf as AX and EDI
// already hold the parameter values:

conv.wToBuf( ax, [edi] );

// The following loads the constant in AX and calls
// conv.wToBuf:

conv.wToBuf( <constant>, [edi] ); // <constant> must fit in 16 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the AX and EDI registers with their respective parameter values. In particular, you should not specify [EAX] as the buffer address because the code that HLA generates can overwrite the L.O. word of EAX (i.e., AX) before it copies the address to the EDI register. It goes without saying that this function will overwrite the values of EAX and EDI if the original parameters are not AX and [EDI].

HLA low-level calling sequence examples:

```
// Passing a word variable and a buffer variable:

mov( wordVariable, ax );
lea( edi, charArrayVariable );
call conv.wToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

mov( wordVariable, ax );
mov( &charArrayVariable, edi );
call conv.wToBuf;

// Passing a pair of registers (that are not
// AX and EDI):

mov( bx, ax );
mov( edx, edi );
call conv.wToBuf;

// Passing a constant:

mov( <constant>, ax );
call conv.wToBuf; // Assume EDI already contains buffer address.
```

**procedure conv.dToBuf( d:dword in eax; var buffer:var in edi );**

Converts the numeric value in EAX to a sequence of exactly eight hexadecimal digits (including leading zeros, as necessary) and stores these characters at the location pointed at by EDI. This function returns EDI pointing at the first byte after the sequence. If the internal underscores flag contains true, this function will emit a nine-character string with a single underscore between the fourth and fifth digits.

HLA high-level calling sequence examples:

```
// The following will load "dwordVariable" into EAX and
// the address of "charArrayVariable" into EDI and then
// call conv.dToBuf:

conv.dToBuf( dwordVariable, charArrayVariable );

// The following call will copy EBX into EAX and
```

```
// EDX into EDI prior to calling conv.dToBuf:

conv.dToBuf( ebx, [edx] );

// The following just calls conv.dToBuf as EAX and EDI
// already hold the parameter values:

conv.dToBuf( eax, [edi] );

// The following loads the constant in EAX and calls
// conv.dToBuf:

conv.dToBuf( <constant>, [edi] ); // <constant> must fit in 32 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the EAX and EDI registers with their respective parameter values. In particular, you should not specify [EAX] as the buffer address because the code that HLA generates can overwrite EAX before it copies the address to the EDI register. It goes without saying that this function will overwrite the values of EAX and EDI if the original parameters are not EAX and [EDI].

HLA low-level calling sequence examples:

```
// Passing a dword variable and a buffer variable:

mov( dwordVariable, eax );
lea( edi, charArrayVariable );
call conv.dToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

mov( dwordVariable, eax );
mov( &charArrayVariable, edi );
call conv.dToBuf;

// Passing a pair of registers (that are not
// EAX and EDI):

mov( ebx, eax );
mov( edx, edi );
call conv.dToBuf;

// Passing a constant:

mov( <constant>, eax );
call conv.dToBuf; // Assume EDI already contains buffer address.
```

**procedure conv.qToBuf( q:qword; var buffer:var in edi );**

Converts the numeric value passed in q to a sequence of exactly 16 hexadecimal digits (including leading zeros, as necessary) and stores these characters at the location pointed at by EDI. This function returns EDI pointing at the first byte after the sequence. If the internal underscores flag contains true, this function will emit a 19-character string with underscores between the 4<sup>th</sup> and 5<sup>th</sup>, 8<sup>th</sup> and 9<sup>th</sup>, and 12<sup>th</sup> and 13<sup>th</sup> digits.

HLA high-level calling sequence examples:

```
// The following will push the value of "qwordVariable"
// onto the stack, load the address of "charArrayVariable"
// into EDI and then call conv.qToBuf:
```

```

conv.qToBuf( qwordVariable, charArrayVariable );

// The following pushes the constant onto the stack and calls
// conv.qToBuf:

conv.qToBuf( <constant>, [edi] ); // <constant> must fit in 64 bits

```

When using the HLA high-level calling form, always keep in mind that these statements load the EDI register with the respective parameter value. It goes without saying that this function will overwrite the value of EDI if the original parameter is not [EDI].

HLA low-level calling sequence examples:

```

// Passing a qword variable and a buffer variable:

    push( (type dword qwordVariable[4])); // H.O. dword first
    push( (type dword qwordVariable[0])); // L.O. dword last
    lea( edi, charArrayVariable );
    call conv.qToBuf;

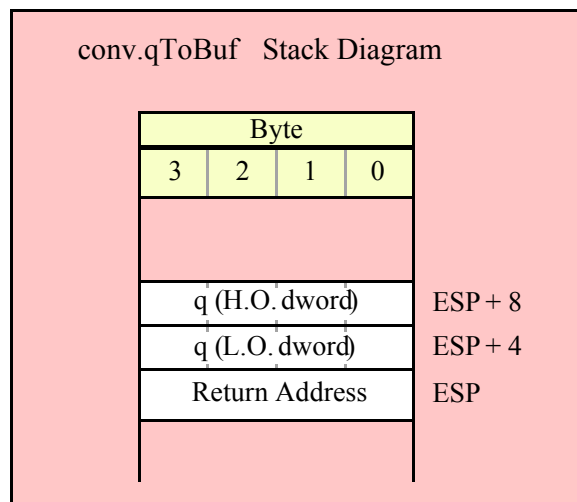
// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

    push( (type dword qwordVariable[4])); // H.O. dword first
    push( (type dword qwordVariable[0])); // L.O. dword last
    mov( &charArrayVariable, edi );
    call conv.qToBuf;

// Passing a constant:

    pushd( <constant> >> 32 ); // Push H.O. dword of constant first.
    pushd( <constant> & $FFFF_FFFF ); // Push L.O. dword second.
    call conv.qToBuf; // Assume EDI already contains buffer address.

```



```
procedure conv.tbToBuf( tb:tbyte; var buffer:var in edi );
```

Converts the numeric value passed in tb to a sequence of exactly 20 hexadecimal digits (including leading zeros, as necessary) and stores these characters at the location pointed at by EDI. This function returns EDI

pointing at the first byte after the sequence. If the internal underscores flag contains true, this function will emit a 24-character string with underscores between the 4<sup>th</sup> and 5<sup>th</sup>, 8<sup>th</sup> and 9<sup>th</sup>, 12<sup>th</sup> and 13<sup>th</sup>, and 16<sup>th</sup> and 17<sup>th</sup> digits.

HLA high-level calling sequence examples:

```
// The following will push the value of "tbyteVariable"
// onto the stack, load the address of "charArrayVariable"
// into EDI and then call conv.tbToBuf:
```

```
conv.tbToBuf( tbyteVariable, charArrayVariable );
```

When using the HLA high-level calling form, always keep in mind that these statements load the EDI register with the respective parameter value. It goes without saying that this function will overwrite the value of EDI if the original parameter is not [EDI].

HLA low-level calling sequence examples:

```
// Passing a tbyte variable and a buffer variable:
```

```
pushw( 0 ); // Must pad parameter to 12 bytes
push( (type word tbyteVariable[8])); // H.O. word first
  push( (type dword tbyteVariable[4]));
  push( (type dword qwordVariable[0])); // L.O. dword last
lea( edi, charArrayVariable );
call conv.tbToBuf;
```

```
// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):
```

```
pushw( 0 ); // Must pad parameter to 12 bytes
push( (type word tbyteVariable[8])); // H.O. word first
  push( (type dword tbyteVariable[4]));
  push( (type dword qwordVariable[0])); // L.O. dword last
mov( &charArrayVariable, edi );
call conv.tbToBuf;
```

```
// Passing a constant:
```

```
pushd( <constant> >> 64 ); // Push H.O. word as dword, first
pushd( ( <constant> >> 32) & $FFFF_FFFF );
pushd( <constant> & $FFFF_FFFF ); // Push L.O. dword last.
call conv.tbToBuf; // Assume EDI already contains buffer address.
```

**procedure conv.lToBuf( l:lword; var buffer:var in edi );**

Converts the numeric value passed in l to a sequence of exactly 32 hexadecimal digits (including leading zeros, as necessary) and stores these characters at the location pointed at by EDI. This function returns EDI pointing at the first byte after the sequence. If the internal underscores flag contains true, this function will emit a 39 character string with underscores between the 4<sup>th</sup> and 5<sup>th</sup>, 8<sup>th</sup> and 9<sup>th</sup>, 12<sup>th</sup> and 13<sup>th</sup>, 16<sup>th</sup> and 17<sup>th</sup>, 20<sup>th</sup> and 21<sup>st</sup>, 24<sup>th</sup> and 25<sup>th</sup>, and 28<sup>th</sup> and 29<sup>th</sup> digits.

HLA high-level calling sequence examples:

```
// The following will push the value of "lwordVariable"
// onto the stack, load the address of "charArrayVariable"
// into EDI and then call conv.lToBuf:
```

```

conv.lToBuf( lwordVariable, charArrayVariable );

// The following pushes the constant onto the stack and calls
// conv.lToBuf:

conv.lToBuf( <constant>, [edi] ); // <constant> must fit in 128 bits

```

When using the HLA high-level calling form, always keep in mind that these statements load the EDI register with the respective parameter value. It goes without saying that this function will overwrite the value of EDI if the original parameter is not [EDI].

HLA low-level calling sequence examples:

```

// Passing an lword variable and a buffer variable:

    push( (type dword lwordVariable[12])); // H.O. dword first
    push( (type dword lwordVariable[8]));
    push( (type dword lwordVariable[4]));
    push( (type dword lwordVariable[0])); // L.O. dword last
    lea( edi, charArrayVariable );
    call conv.lToBuf;

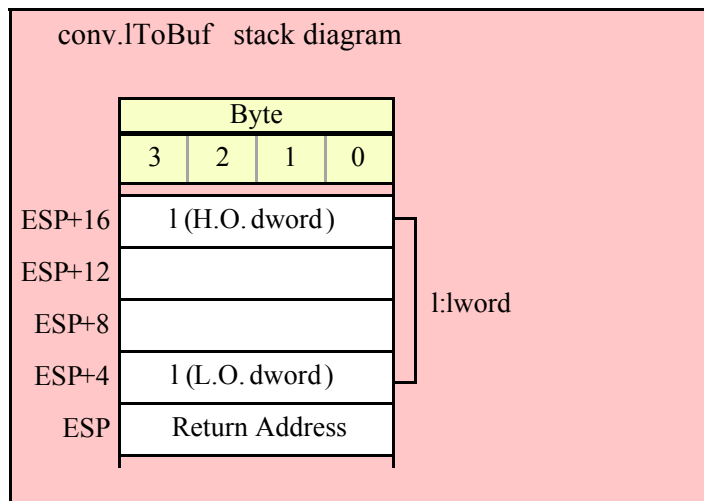
// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

    push( (type dword lwordVariable[12])); // H.O. dword first
    push( (type dword lwordVariable[8]));
    push( (type dword lwordVariable[4]));
    push( (type dword lwordVariable[0])); // L.O. dword last
    mov( &charArrayVariable, edi );
    call conv.lToBuf;

// Passing a constant:

    pushd( <constant> >> 96 ); // Push H.O. dword of constant first.
    pushd( (<constant> >> 64) & $FFFF_FFFF );
    pushd( (<constant> >> 32) & $FFFF_FFFF );
    pushd( <constant> & $FFFF_FFFF ); // Push L.O. dword last.
    call conv.lToBuf; // Assume EDI already contains buffer address.

```



### 8.3.3.2 Variable Length Hexadecimal Numeric to Buffer Conversions

The variable length hexadecimal to buffer conversion routines translate an input numeric value to a variable-length string (depending on the value, data type size, and the setting of the internal underscores flag). These functions emit the characters of the string (without leading zeros) to sequential locations starting at the address held in EDI.

```
procedure conv.h8ToBuf( b:byte in al; var buffer:var in edi );
```

Converts the numeric value in AL to a sequence of one or two hexadecimal digits and stores these two characters at the location pointed at by EDI. This function returns EDI pointing at the first byte after the sequence. Because the conversion is always less than five digits, the internal underscores flag does not affect the output this function produces.

HLA high-level calling sequence examples:

```
// The following will load "byteVariable" into AL and
// the address of "charArrayVariable" into EDI and then
// call conv.h8ToBuf:

conv.h8ToBuf( byteVariable, charArrayVariable );

// The following call will copy BH into AL and
// EDX into EDI prior to calling conv.h8ToBuf:

conv.h8ToBuf( bh, [edx] );

// The following just calls conv.h8ToBuf as AL and EDI
// already hold the parameter values:

conv.h8ToBuf( al, [edi] );

// The following loads the constant in AL and calls
// conv.h8ToBuf:

conv.h8ToBuf( <constant>, [edi] ); // <constant> must fit in 8 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the AL and EDI registers with their respective parameter values. In particular, you should not specify [EAX] as the buffer address because the code that HLA generates can overwrite the L.O. byte of EAX (i.e., AL) before it copies the address to the EDI register. It goes without saying that this function will overwrite the values of EAX and EDI if the original parameters are not AL and [EDI].

HLA low-level calling sequence examples:

```
// Passing a byte variable and a buffer variable:

mov( byteVariable, al );
lea( edi, charArrayVariable );
call conv.h8ToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

mov( byteVariable, al );
mov( &charArrayVariable, edi );
call conv.h8ToBuf;

// Passing a pair of registers (that are not
// AL and EDI):
```



```

mov( bh, al );
mov( edx, edi );
call conv.h8ToBuf;

// Passing a constant:

mov( <constant>, al );
call conv.h8ToBuf; // Assume EDI already contains buffer address.

```

**procedure conv.h16ToBuf( w:word in ax; var buffer:var in edi );**

Converts the numeric value in AX to a sequence of one to four hexadecimal digits and stores these characters at the location pointed at by EDI. This function returns EDI pointing at the first byte after the sequence. Because the conversion is always less than five digits, the internal underscores flag does not affect the output this function produces.

HLA high-level calling sequence examples:

```

// The following will load "wordVariable" into AX and
// the address of "charArrayVariable" into EDI and then
// call conv.h16ToBuf:

conv.h16ToBuf( wordVariable, charArrayVariable );

// The following call will copy BX into AX and
// EDX into EDI prior to calling conv.h16ToBuf:

conv.h16ToBuf( bx, [edx] );

// The following just calls conv.h16ToBuf as AX and EDI
// already hold the parameter values:

conv.h16ToBuf( ax, [edi] );

// The following loads the constant in AX and calls
// conv.h16ToBuf:

conv.h16ToBuf( <constant>, [edi] ); // <constant> must fit in 16 bits

```

When using the HLA high-level calling form, always keep in mind that these statements load the AX and EDI registers with their respective parameter values. In particular, you should not specify [EAX] as the buffer address because the code that HLA generates can overwrite the L.O. word of EAX (i.e., AX) before it copies the address to the EDI register. It goes without saying that this function will overwrite the values of EAX and EDI if the original parameters are not AX and [EDI].

HLA low-level calling sequence examples:

```

// Passing a word variable and a buffer variable:

mov( wordVariable, ax );
lea( edi, charArrayVariable );
call conv.h16ToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

mov( wordVariable, ax );
mov( &charArrayVariable, edi );
call conv.h16ToBuf;

```

```
// Passing a pair of registers (that are not
// AX and EDI):

mov( bx, ax );
mov( edx, edi );
call conv.h16ToBuf;

// Passing a constant:

mov( <constant>, ax );
call conv.h16ToBuf; // Assume EDI already contains buffer address.
```

**procedure conv.h32ToBuf( d:dword in eax; var buffer:var in edi );**

Converts the numeric value in EAX to a sequence of one to eight hexadecimal digits and stores these characters at the location pointed at by EDI. This function returns EDI pointing at the first byte after the sequence. If the internal underscores flag contains true and the value is \$1\_0000 or greater, then this function will emit an underscore between the fourth and fifth digits in the output string.

HLA high-level calling sequence examples:

```
// The following will load "dwordVariable" into EAX and
// the address of "charArrayVariable" into EDI and then
// call conv.h32ToBuf:

conv.h32ToBuf( dwordVariable, charArrayVariable );

// The following call will copy EBX into EAX and
// EDX into EDI prior to calling conv.h32ToBuf:

conv.h32ToBuf( ebx, [edx] );

// The following just calls conv.h32ToBuf as EAX and EDI
// already hold the parameter values:

conv.h32ToBuf( eax, [edi] );

// The following loads the constant in EAX and calls
// conv.h32ToBuf:

conv.h32ToBuf( <constant>, [edi] ); // <constant> must fit in 32 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the EAX and EDI registers with their respective parameter values. In particular, you should not specify [EAX] as the buffer address because the code that HLA generates can overwrite EAX before it copies the address to the EDI register. It goes without saying that this function will overwrite the values of EAX and EDI if the original parameters are not EAX and [EDI].

HLA low-level calling sequence examples:

```
// Passing a dword variable and a buffer variable:

mov( dwordVariable, eax );
lea( edi, charArrayVariable );
call conv.h32ToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):
```

```

mov( dwordVariable, eax );
mov( &charArrayVariable, edi );
call conv.h32ToBuf;

// Passing a pair of registers (that are not
// EAX and EDI):

mov( ebx, eax );
mov( edx, edi );
call conv.h32ToBuf;

// Passing a constant:

mov( <constant>, eax );
call conv.h32ToBuf; // Assume EDI already contains buffer address.

```

**procedure conv.h64ToBuf( q:qword; var buffer:var in edi );**

Converts the numeric value in q to a sequence of 1 to 16 hexadecimal digits and stores these characters at the location pointed at by EDI. This function returns EDI pointing at the first byte after the sequence.

If the internal underscores flag contains true and the value is \$1\_0000 or greater, then this function will emit an underscore between the 4<sup>th</sup> and 5<sup>th</sup> digits in the output string.

If the value is \$1\_0000 or greater, then this function will emit an underscore between the 4<sup>th</sup> and 5<sup>th</sup> digits in the output string.

If the value is \$1\_0000\_0000 or greater, then this function will emit an underscore between the 8<sup>th</sup> and 9<sup>th</sup> digits.

If the value is \$1\_0000\_0000\_0000 or greater, then this function will emit an underscore between the 12<sup>th</sup> and 13<sup>th</sup> digits.

HLA high-level calling sequence examples:

```

// The following will push the value of "qwordVariable"
// onto the stack, load the address of "charArrayVariable"
// into EDI and then call conv.h64ToBuf:

conv.h64ToBuf( qwordVariable, charArrayVariable );

// The following pushes the constant onto the stack and calls
// conv.h64ToBuf:

conv.h64ToBuf( <constant>, [edi] ); // <constant> must fit in 64 bits

```

When using the HLA high-level calling form, always keep in mind that these statements load the EDI register with the respective parameter value. It goes without saying that this function will overwrite the value of EDI if the original parameter is not [EDI].

HLA low-level calling sequence examples:

```

// Passing a qword variable and a buffer variable:

push( (type dword qwordVariable[4])); // H.O. dword first
push( (type dword qwordVariable[0])); // L.O. dword last
lea( edi, charArrayVariable );
call conv.h64ToBuf;

// Alternate form of above if charArrayVariable is

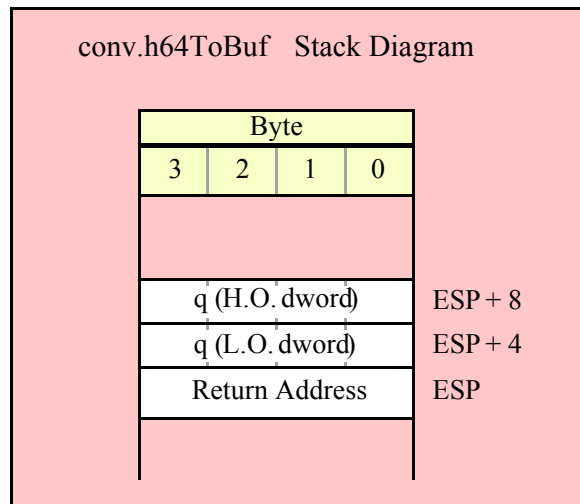
```

```
// a static object (STATIC, READONLY, or STORAGE):

    push( (type dword qwordVariable[4])); // H.O. dword first
    push( (type dword qwordVariable[0])); // L.O. dword last
    mov( &charArrayVariable, edi );
    call conv.h64ToBuf;

// Passing a constant:

    pushd( <constant> >> 32 ); // Push H.O. dword of constant first.
    pushd( <constant> & $FFFF_FFFF ); // Push L.O. dword second.
    call conv.h64ToBuf; // Assume EDI already contains buffer address.
```



```
procedure conv.h80ToBuf( tb:tbyte; var buffer:var in edi );
```

Converts the numeric value in tb to a sequence of 1 to 20 hexadecimal digits and stores these characters at the location pointed at by EDI. This function returns EDI pointing at the first byte after the sequence.

If the internal underscores flag contains true, and:

If the value is \$1\_0000 or greater, then this function will emit an underscore between the 4<sup>th</sup> and 5<sup>th</sup> digits in the output string.

If the value is \$1\_0000\_0000 or greater, then this function will emit an underscore between the 8<sup>th</sup> and 9<sup>th</sup> digits.

If the value is \$1\_0000\_0000\_0000 or greater, then this function will emit an underscore between the 12<sup>th</sup> and 13<sup>th</sup> digits.

If the value is \$1\_0000\_0000\_0000\_0000 or greater, then this function will emit an underscore between the 16<sup>th</sup> and 17<sup>th</sup> digits.

HLA high-level calling sequence examples:

```
// The following will push the value of "tbyteVariable"
// onto the stack, load the address of "charArrayVariable"
// into EDI and then call conv.h80ToBuf:

conv.h80ToBuf( tbyteVariable, charArrayVariable );
```

When using the HLA high-level calling form, always keep in mind that these statements load the EDI register with the respective parameter value. It goes without saying that this function will overwrite the value of EDI if the original parameter is not [EDI].

HLA low-level calling sequence examples:

```
// Passing a tbyte variable and a buffer variable:

pushw( 0 ); // Must pad parameter to 12 bytes
push( (type word tbyteVariable[8])); // H.O. word first
  push( (type dword tbyteVariable[4]));
  push( (type dword qwordVariable[0])); // L.O. dword last
lea( edi, charArrayVariable );
call conv.h80ToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

pushw( 0 ); // Must pad parameter to 12 bytes
push( (type word tbyteVariable[8])); // H.O. word first
  push( (type dword tbyteVariable[4]));
  push( (type dword qwordVariable[0])); // L.O. dword last
mov( &charArrayVariable, edi );
call conv.h80ToBuf;

// Passing a constant:

pushd( <constant> >> 64 ); // Push H.O. word as dword, first
pushd( (<constant> >> 32) & $FFFF_FFFF );
pushd( <constant> & $FFFF_FFFF ); // Push L.O. dword last.
call conv.tbToBuf; // Assume EDI already contains buffer address.
```

**procedure conv.h128ToBuf( l:lword; var buffer:var in edi );**

Converts the numeric value in l to a sequence of 1 to 32 hexadecimal digits and stores these characters at the location pointed at by EDI. This function returns EDI pointing at the first byte after the sequence.

If the internal underscores flag contains true, and:

If the value is \$1\_0000 or greater, then this function will emit an underscore between the 4<sup>th</sup> and 5<sup>th</sup> digits in the output string.

If the value is \$1\_0000\_0000 or greater, then this function will emit an underscore between the 8<sup>th</sup> and 9<sup>th</sup> digits.

If the value is \$1\_0000\_0000\_0000 or greater, then this function will emit an underscore between the 12<sup>th</sup> and 13<sup>th</sup> digits.

If the value is \$1\_0000\_0000\_0000\_0000 or greater, then this function will emit an underscore between the 16<sup>th</sup> and 17<sup>th</sup> digits.

If the value is \$1\_0000\_0000\_0000\_0000\_0000 or greater, then this function will emit an underscore between the 20<sup>th</sup> and 21<sup>st</sup> digits.

If the value is \$1\_0000\_0000\_0000\_0000\_0000\_0000 or greater, then this function will emit an underscore between the 24<sup>th</sup> and 25<sup>th</sup> digits.

If the value is \$1\_0000\_0000\_0000\_0000\_0000\_0000\_0000 or greater, then this function will emit an underscore between the 28<sup>th</sup> and 29<sup>th</sup> digits.

HLA high-level calling sequence examples:

```
// The following will push the value of "lwordVariable"
// onto the stack, load the address of "charArrayVariable"
// into EDI and then call conv.h128ToBuf:
```

```
conv.h128ToBuf( lwordVariable, charArrayVariable );
```

```
// The following pushes the constant onto the stack and calls
// conv.h128ToBuf:

conv.h128ToBuf( <constant>, [edi] ); // <constant> must fit in 128 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the EDI register with the respective parameter value. It goes without saying that this function will overwrite the value of EDI if the original parameter is not [EDI].

HLA low-level calling sequence examples:

```
// Passing an lword variable and a buffer variable:

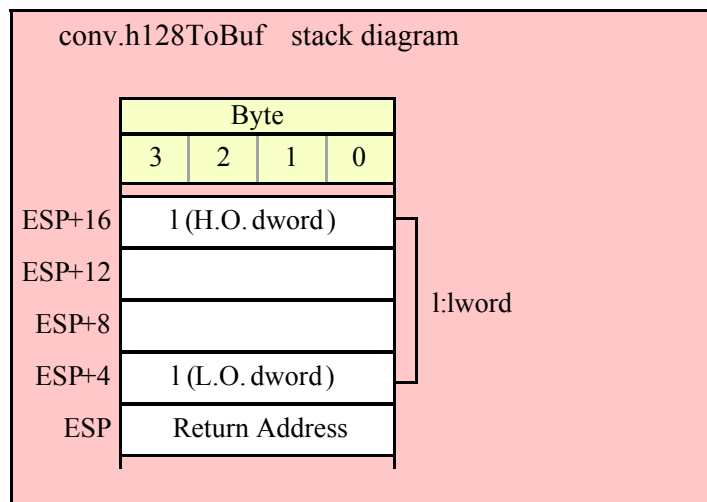
push( (type dword lwordVariable[12])); // H.O. dword first
push( (type dword lwordVariable[8]));
push( (type dword lwordVariable[4]));
push( (type dword lwordVariable[0])); // L.O. dword last
lea( edi, charArrayVariable );
call conv.h128ToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

push( (type dword lwordVariable[12])); // H.O. dword first
push( (type dword lwordVariable[8]));
push( (type dword lwordVariable[4]));
push( (type dword lwordVariable[0])); // L.O. dword last
mov( &charArrayVariable, edi );
call conv.h128ToBuf;

// Passing a constant:

pushd( <constant> >> 96 ); // Push H.O. dword of constant first.
pushd( (<constant> >> 64) & $FFFF_FFFF );
pushd( (<constant> >> 32) & $FFFF_FFFF );
pushd( <constant> & $FFFF_FFFF ); // Push L.O. dword last.
call conv.h128ToBuf; // Assume EDI already contains buffer address.
```



### 8.3.4 Hexadecimal Numeric to String Conversions

The hexadecimal numeric to string conversion routines are the general-purpose hexadecimal string conversion routines. There are two versions: one set of these routines store their string data into a preallocated string object (the *unadorned* versions), the other set (*adorned* with an "a\_" prefix in their name) allocates storage for a new string on the heap and returns a pointer to that new string in the EAX register.

As for the buffer routines (e.g., `conv.bToBuf` and `conv.h8ToBuf`) there are two categories of routines based on whether the routines emit a minimum length string or pad the string with leading zeros to the data type's *natural* size. The `conv.hXToStr` routines emit the minimum number of hexadecimal digits in the string they create.

The `conv.XToStr` ( $X = b, w, d, q, tb, \text{ or } l$ ) functions always create a fixed length string with an appropriate number of leading zeros ( $b=2$  digits,  $w=4$  digits,  $d=8$  digits,  $q=16$  digits,  $tb=20$  digits, and  $l=32$  digits). If the internal underscores flag contains true, then these functions will emit an underscore between each group of four hexadecimal digits.

The `conv.hXToStr` functions ( $X=8, 16, 32, 64, 80, \text{ or } 128$ ) let you specify a minimum field width and a padding character (because the `conv.XToBuf` routines always emit fixed-length strings, there is no need to specify a minimum field width or padding character for those strings). The absolute value of the width parameter specifies the minimum string length for the conversion. The conversion will always produce a string at least `abs(width)` characters long. If the conversion would require more than `abs(width)` print positions, then the conversion will produce a larger string, as necessary.

If the string conversion requires fewer than `abs(width)` characters and the width parameter is a non-negative value, these routines right justify the value in the string and pad the remaining positions with the fill character. If the conversion requires fewer than `abs(width)` characters and the width parameter is a negative number, then these functions will left-justify the value in the output field and pad the end of the string with the fill character.

`xxxToStr ( value, width, fill, buffer );`

Assuming "value" requires five print positions, "width" is eight, and fill is "f" then the `xxxToStr` functions produce the string

f	f	f	V	A	L	U	E
---	---	---	---	---	---	---	---

Assuming "value" requires five print positions, "width" is minus eight, and fill is "f" then the `xxxToStr` functions produce the string

V	A	L	U	E	f	f	f
---	---	---	---	---	---	---	---

For the unadorned functions, the destination string's maximum length must be large enough to hold the full result (including any extra print positions needed beyond the value specified by `abs(width)`) or these functions will raise a string overflow exception.

If the internal underscore flag is true, then the 32-bit and larger hex to string conversion functions will emit an underscore between each set of four hexadecimal digits, starting from the least significant digit. This is true for both the `conv.hXToStr` and `conv.XToStr` routines. See the descriptions of the `conv.setUnderscores` and `conv.getUnderscores` functions for more details. Note that the hexadecimal to numeric conversion functions do not inject underscores into sequences of padding characters, only into the actual digits the conversion produces. This is true even if you specify a numeric character (such as '0') as the padding character.

#### 8.3.4.1 Fixed-Length Numeric to Hexadecimal String Conversions

The functions in this category convert an 8-bit, 16-bit, 32-bit, 64-bit, 80-bit, or 128-bit numeric value to fixed-length strings (one hexadecimal digit for each four bits, including leading zeros). If the internal underscores flag contains true, then these functions will also insert an underscore between each group of four hexadecimal digits.

`conv.bToStr( b:byte; dest:string );`

This function converts the 8-bit value of the `b` parameter to a two-byte string containing `b`'s hexadecimal representation. Because this conversion requires exactly two digits, the internal underscores flag setting does not affect the operation of this function. This function raises a string overflow error if the destination string is not large enough to hold the converted string.

HLA high-level calling sequence examples:

```
// The following will push "byteVariable" and "destStr" (which
// is a pointer to a string object) and then call conv.bToStr:

conv.bToStr( byteVariable, destStr );

// The following call will BH's value onto the stack and then
// push EDX's value (which is the address of a string object)
// before calling conv.bToStr:

conv.bToStr( bh, edx );

// The following pushes the constant and destStr and calls
// conv.bToStr:

conv.bToBuf( <constant>, destStr ); // <constant> must fit in 8 bits
```

When using the HLA high-level calling form, remember that string variables are dword pointers that contain the address of a string object. The destination string parameter is passed by value, not by reference; it just turns out that the value of a string is a pointer to the actual string data (confusing, isn't it?). In any case, this is why you can pass a register value as the destination string location rather than having to pass something like "[edx]". A construct like "[edx]" would imply that EDX contains the address of the string variable, that is, a pointer to the pointer to the string.

HLA low-level calling sequence examples:

```
// Passing a byte variable and a buffer variable, option 1
// (if a 32-bit register is available):

movzx( byteVariable, eax );
push( eax );
push( destStr );
call conv.bToStr;

// Passing a byte variable and a buffer variable, option 2
// (if byteVariable isn't the last byte in mapped memory):

push( (type dword byteVariable));
push( destStr );
call conv.bToStr;

// Passing a byte variable and a buffer variable, option 3
// No registers are available and we can't guarantee that
// the three bytes following byteVariable are present in
// mapped memory:

sub( 4, esp );
push( eax );
movzx( byteVariable, eax );
mov( eax, [esp+4] );
pop( eax );
push( destStr );
call conv.bToStr;

// Passing a pair of registers (hex value in AL, BL, CL, or DL):
// BL = value to print, EDX = pointer to string object.

push( ebx );      // Pushes BL
push( edx );
call conv.bToStr;
```

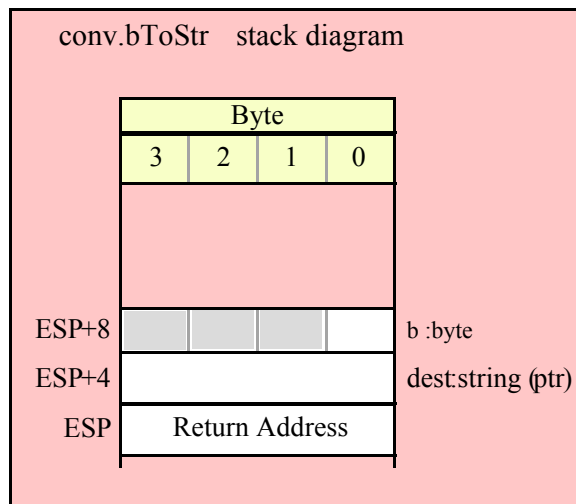


```
// Passing a pair of registers (hex value in AH, BH, CH, or DH):
// BH = value to print, EDX = pointer to string object.

pushd( 0 );
mov( bh, [esp] );
push( edx );
call conv.bToStr;

// Passing a constant:

pushd( <constant> );
push( destStr );
call conv.bToStr;
```



```
conv.a_bToStr( b:byte ); @returns( "eax" );
```

This function converts the value of the `b` parameter to a two-byte string containing `b`'s hexadecimal representation. Because this conversion requires exactly two digits, the internal underscores flag setting does not affect the operation of this function. This function allocates storage for the string on the heap and returns a pointer to that string in the `EAX` register.

HLA high-level calling sequence examples:

```
// The following will push "byteVariable" and then call conv.a_bToStr:
```

```
conv.a_bToStr( byteVariable );
mov( eax, byteStr );
```

```
// The following call will BH's value onto the stack
// before calling conv.a_bToStr:
```

```
conv.a_bToStr( bh );
mov( eax, byteStr );
```

```
// The following pushes the constant and calls
// conv.a_bToStr:
```

```
conv.bToBuf( <constant> ); // <constant> must fit in 8 bits
mov( eax, byteStr );
```

HLA low-level calling sequence examples:

```
// Passing a byte variable and a buffer variable, option 1
// (if a 32-bit register is available):

movzx( byteVariable, eax );
push( eax );
call conv.a_bToStr;
mov( eax, destStr );

// Passing a byte variable and a buffer variable, option 2
// (if byteVariable isn't the last byte in mapped memory):

push( (type dword byteVariable));
call conv.a_bToStr;
mov( eax, destStr );

// Passing a byte variable, option 3
// No registers are available and we can't guarantee that
// the three bytes following byteVariable are present in
// mapped memory:

sub( 4, esp );
push( eax );
movzx( byteVariable, eax );
mov( eax, [esp+4] );
pop( eax );
call conv.a_bToStr;
mov( eax, destStr );

// Passing a pair of registers (hex value in AL, BL, CL, or DL):
// BL = value to print, EDX = pointer to string object.

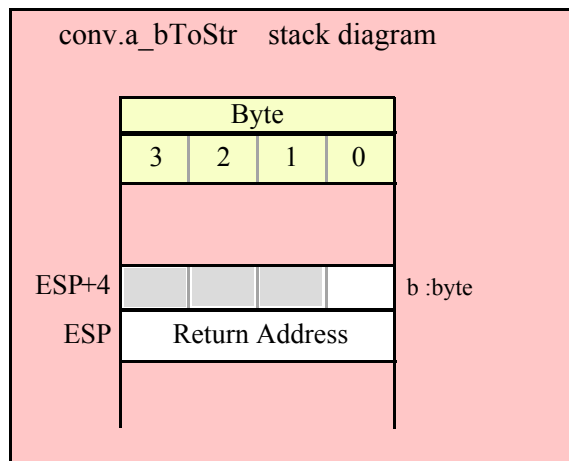
push( ebx );          // Pushes BL
call conv.a_bToStr;
mov( eax, byteStr );

// Passing a pair of registers (hex value in AH, BH, CH, or DH):
// BH = value to print, EDX = pointer to string object.

pushd( 0 );
mov( bh, [esp] );
call conv.a_bToStr;
mov( eax, byteStr );

// Passing a constant:

pushd( <constant> );
call conv.a_bToStr;
mov( eax, byteStr );
```



```
conv.wToStr( w:word; dest:string );
```

Converts the 16-bit value of the `w` parameter to a four-byte string that is the hexadecimal representation of this value. Because this conversions requires exactly four digits, the internal underscores flag setting does not affect the operation of this function. This function raises a string overflow error if the destination string is not large enough to hold the converted string.

HLA high-level calling sequence examples:

```
// The following will push "wordVariable" and "destStr" (which
// is a pointer to a string object) and then call conv.wToStr:
```

```
conv.wToStr( wordVariable, destStr );
```

```
// The following call will BX's value onto the stack and then
// push EDX's value (which is the address of a string object)
// before calling conv.wToStr:
```

```
conv.wToStr( bx, edx );
```

```
// The following pushes the constant and destStr and calls
// conv.wToStr:
```

```
conv.wToStr( <constant>, destStr ); // <constant> must fit in 16 bits
```

When using the HLA high-level calling form, remember that string variables are dword pointers that contain the address of a string object. The destination string parameter is passed by value, not by reference; it just turns out that the value of a string is a pointer to the actual string data (confusing, isn't it?). In any case, this is why you can pass a register value as the destination string location rather than having to pass something like "[edx]". A construct like "[edx]" would imply that EDX contains the address of the string variable, that is, a pointer to the pointer to the string.

HLA low-level calling sequence examples:

```
// Passing a word variable and a buffer variable, option 1
// (if a 32-bit register is available):
```

```
movzx( byteVariable, eax );
push( eax );
push( destStr );
call conv.wToStr;
```

```

// Passing a word variable and a buffer variable, option 2
// (if wordVariable isn't the last byte in mapped memory):

push( (type dword wordVariable));
push( destStr );
call conv.wToStr;

// Passing a word variable and a buffer variable, option 3
// No registers are available and we can't guarantee that
// the two bytes following wordVariable are present in
// mapped memory:

pushw( 0 );
push( wordVariable );
push( destStr );
call conv.wToStr;

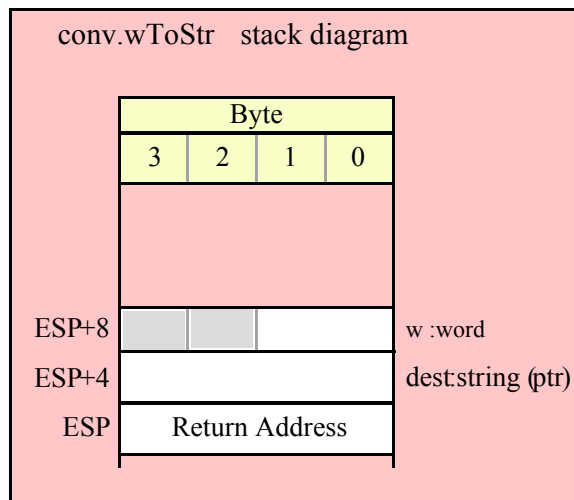
// Passing a pair of registers:
// BX = value to print, EDX = pointer to string object.

push( ebx );      // Pushes BX
push( edx );
call conv.wToStr;

// Passing a constant:

pushd( <constant> );
push( destStr );
call conv.wToStr;

```



```
conv.a_wToStr( w:word; dest:string ); @returns( "eax" );
```

Converts the 16-bit value of the `w` parameter to a four-byte string that is the hexadecimal representation of this value. Because this conversions requires exactly four digits, the internal underscores flag setting does not affect the operation of this function. This function allocates storage for the string on the heap and returns a pointer to that string in the EAX register.

HLA high-level calling sequence examples:

```
// The following will push "wordVariable" and then call conv.wToStr:
```

```

conv.a_wToStr( wordVariable );
mov( eax, destStr );

// The following call will BX's value onto the stack
// before calling conv.a_wToStr:

conv.a_wToStr( bx );

// The following pushes the constant and calls
// conv.a_wToStr:

conv.a_wToStr( <constant>, destStr ); // <constant> must fit in 16 bits

```

HLA low-level calling sequence examples:

```

// Passing a word variable and a buffer variable, option 1
// (if a 32-bit register is available):

```

```

movzx( wordVariable, eax );
push( eax );
call conv.a_wToStr;
mov( eax, destStr );

```

```

// Passing a word variable and a buffer variable, option 2
// (if wordVariable isn't the last byte in mapped memory):

```

```

push( (type dword wordVariable));
call conv.wToStr;
mov( eax, destStr );

```

```

// Passing a word variable and a buffer variable, option 3
// No registers are available and we can't guarantee that
// the two bytes following wordVariable are present in
// mapped memory:

```

```

pushw( 0 );
push( wordVariable );
call conv.wToStr;
mov( eax, destStr );

```

```

// Passing a pair of registers:
// BX = value to print.

```

```

push( ebx );          // Pushes BX
call conv.wToStr;
mov( eax, wordStr );

```

```

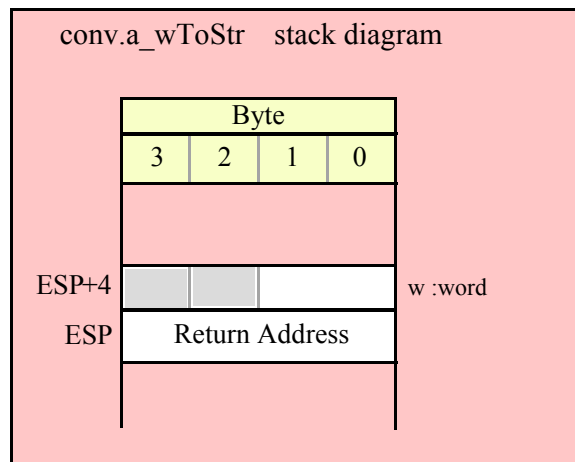
// Passing a constant:

```

```

pushd( <constant> );
call conv.wToStr;
mov( eax, destStr );

```



```
conv.dToStr( d:dword; dest:string );
```

This function converts the 32-bit value of the `d` parameter to an eight- or nine-byte string that is the hexadecimal representation of this value. If `d`'s value is greater than \$FFFF and the underscores flag is true, then this function emits a nine-character string with an underscore between the fourth and fifth digits (counting from the least significant digit). This function raises a string overflow error if the destination string is not large enough to hold the converted string.

HLA high-level calling sequence examples:

```
// The following will push "dwordVariable" and "destStr" (which
// is a pointer to a string object) and then call conv.dToStr:
```

```
conv.dToStr( dwordVariable, destStr );
```

```
// The following call will push EBX's value onto the stack and then
// push EDX's value (which is the address of a string object)
// before calling conv.dToStr:
```

```
conv.dToStr( ebx, edx );
```

```
// The following pushes the constant and destStr and calls
// conv.dToStr:
```

```
conv.dToStr( <constant>, destStr ); // <constant> must fit in 32 bits
```

When using the HLA high-level calling form, remember that string variables are dword pointers that contain the address of a string object. The destination string parameter is passed by value, not by reference; it just turns out that the value of a string is a pointer to the actual string data (confusing, isn't it?). In any case, this is why you can pass a register value as the destination string location rather than having to pass something like "[edx]". A construct like "[edx]" would imply that EDX contains the address of the string variable, that is, a pointer to the pointer to the string.

HLA low-level calling sequence examples:

```
// Passing a dword variable and a buffer variable:
```

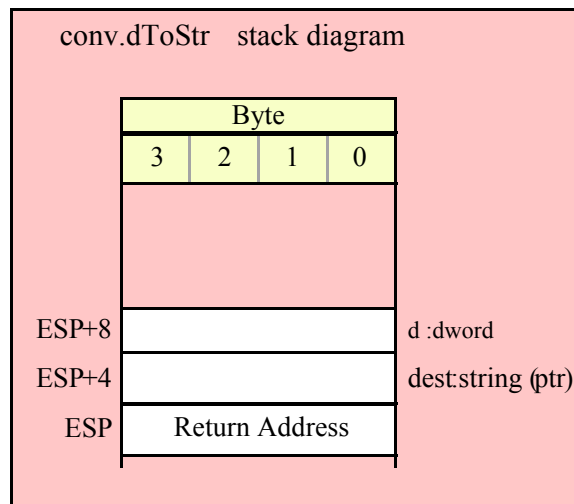
```
push( dwordVariable );
push( destStr );
call conv.dToStr;
```

```
// Passing a pair of registers:
// EBX = value to print, EDX = pointer to string object.

push( ebx );
push( edx );
call conv.dToStr;

// Passing a constant:

pushd( <constant> );
push( destStr );
call conv.dToStr;
```



```
conv.a_dToStr( d:dword; dest:string ); @returns( "eax" );
```

Converts the 32-bit value of the `d` parameter to an eight- or nine-byte string that is the hexadecimal representation of this value. If `d`'s value is greater than \$FFFF and the underscores flag is true, then this function emits a nine-character string with an underscore between the fourth and fifth digits (counting from the least significant digit). This function allocates storage for the string on the heap and returns a pointer to that string in the EAX register.

HLA high-level calling sequence examples:

```
// The following will push "dwordVariable" and then call conv.a_dToStr:
```

```
conv.a_dToStr( dwordVariable );
mov( eax, destStr );
```

```
// The following call will push EBX's value onto the stack
// before calling conv.a_dToStr:
```

```
conv.a_dToStr( ebx );
```

```
// The following pushes the constant and calls
// conv.a_dToStr:
```

```
conv.a_dToStr( <constant>, destStr ); // <constant> must fit in 32 bits
```

HLA low-level calling sequence examples:

```
// Passing a dword variable:
```

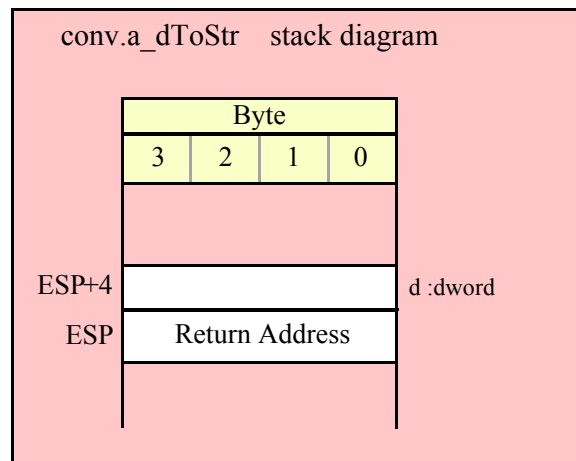
```
push( dwordVariable );
call conv.a_dToStr;
mov( eax, destStr );
```

```
// Passing a register:
// EBX = value to print.
```

```
push( ebx );
call conv.a_dToStr;
mov( eax, dwordStr );
```

```
// Passing a constant:
```

```
pushd( <constant> );
call conv.a_dToStr;
mov( eax, destStr );
```



```
conv.qToStr( q:qword; dest:string );
```

Converts the 64-bit value of the `q` parameter to 16- or 19-byte string (based on the setting of the underscores flag) that is the hexadecimal representation of this value. If the internal underscores flag is true, then this function emits an underscore between each group of four hexadecimal digits (resulting in a 19-character string). This function raises a string overflow error if the destination string is not large enough to hold the converted string.

HLA high-level calling sequence examples:

```
// The following will push the value of "qwordVariable"
// and the value of the destStr string variable
// onto the stack and then call conv.qToStr:
```

```
conv.qToStr( qwordVariable, destStr );
```



```
// The following pushes the constant onto the stack along with
// the value held in the destStr variable and calls
// conv.qToStr:

conv.qToStr( <constant>, destStr ); // <constant> must fit in 64 bits
```

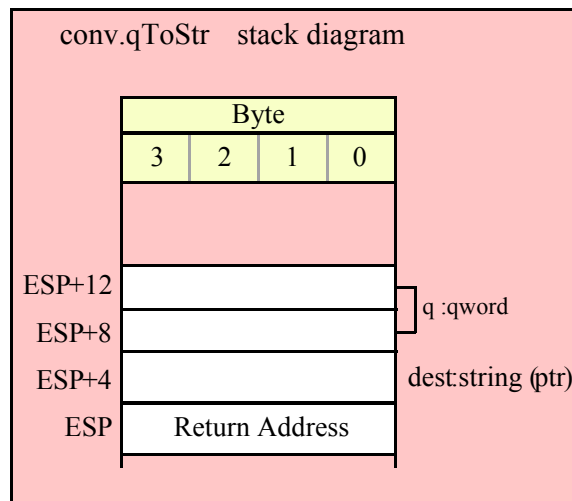
HLA low-level calling sequence examples:

```
// Passing a qword variable and a buffer variable:
```

```
    push( (type dword qwordVariable[4])); // H.O. dword first
    push( (type dword qwordVariable[0])); // L.O. dword last
push( destStr );
call conv.qToStr;
```

```
// Passing a constant:
```

```
pushd( <constant> >> 32 ); // Push H.O. dword of constant first.
pushd( <constant> & $FFFF_FFFF ); // Push L.O. dword second.
push( destStr );
call conv.qToStr;
```



```
conv.a_qToStr( q:qword; dest:string ); @returns( "eax" );
```

Converts the 64-bit value of the `q` parameter to 16- or 19-byte string (based on the setting of the underscores flag) that is the hexadecimal representation of this value. If the internal underscores flag is true, then this function emits an underscore between each group of four hexadecimal digits (resulting in a 19-character string). This function allocates storage for the string on the heap and returns a pointer to that string in the EAX register.

HLA high-level calling sequence examples:

```
// The following will push the value of "qwordVariable"
// onto the stack and then call conv.a_qToStr:
```

```
conv.a_qToStr( qwordVariable );
mov( eax, destStr );
```

```
// The following pushes the constant onto the stack and calls
// conv.a_qToStr:

conv.a_qToStr( <constant> ); // <constant> must fit in 64 bits
mov( eax, destStr );
```

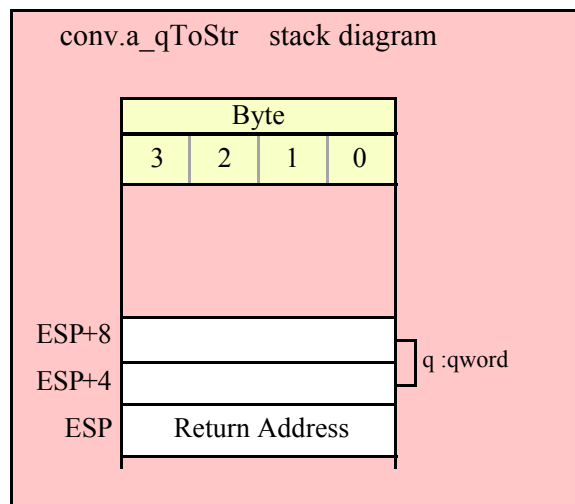
HLA low-level calling sequence examples:

```
// Passing a qword variable:
```

```
    push( (type dword qwordVariable[4])); // H.O. dword first
    push( (type dword qwordVariable[0])); // L.O. dword last
call conv.a_qToStr;
mov( eax, destStr );
```

```
// Passing a constant:
```

```
pushd( <constant> >> 32 ); // Push H.O. dword of constant first.
pushd( <constant> & $FFFF_FFFF ); // Push L.O. dword second.
call conv.a_qToStr;
mov( eax, destStr );
```



```
conv.tbToStr( tb:tbyte; dest:string );
```

Converts the 80-bit value of the `d` parameter to 20- or 24-byte string (based on the setting of the underscores flag) that is the hexadecimal representation of this value. If the internal underscores flag is true, then this function emits an underscore between each group of four hexadecimal digits (resulting in a 24-character string). This function raises a string overflow error if the destination string is not large enough to hold the converted string.

HLA high-level calling sequence examples:

```
// The following will push the value of "tbyteVariable"
// and the value of destStr onto the stack
// and then call conv.tbToStr:
```

```
conv.tbToStr( tbyteVariable, destStr );
```

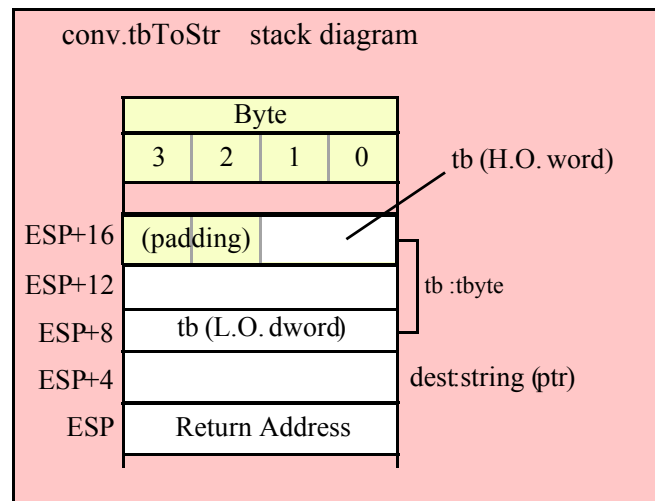
HLA low-level calling sequence examples:

```
// Passing a tbyte variable:
```

```
pushw( 0 ); // Must pad parameter to 12 bytes
push( (type word tbyteVariable[8])); // H.O. word first
  push( (type dword tbyteVariable[4]));
  push( (type dword qwordVariable[0])); // L.O. dword last
push( destStr );
call conv.tbToStr;
```

```
// Passing a constant:
```

```
pushd( <constant> >> 64 ); // Push H.O. word as dword, first
pushd( (<constant> >> 32) & $FFFF_FFFF );
pushd( <constant> & $FFFF_FFFF ); // Push L.O. dword last.
push( destStr );
call conv.tbToStr;
```



```
conv.a_tbToStr( tb:tbyte; dest:string ); @returns( "eax" );
```

Converts the 80-bit value of the d parameter to 20- or 24-byte string (based on the setting of the underscores flag) that is the hexadecimal representation of this value. If the internal underscores flag is true, then this function emits an underscore between each group of four hexadecimal digits (resulting in a 24-character string). This function allocates storage for the string on the heap and returns a pointer to that string in the EAX register.

HLA high-level calling sequence examples:

```
// The following will push the value of "tbyteVariable"
// onto the stack and then call conv.a_tbToStr:
```

```
conv.a_tbToStr( tbyteVariable );
mov( eax, destStr );
```

HLA low-level calling sequence examples:

```
// Passing a tbyte variable:
```

```
pushw( 0 ); // Must pad parameter to 12 bytes
```

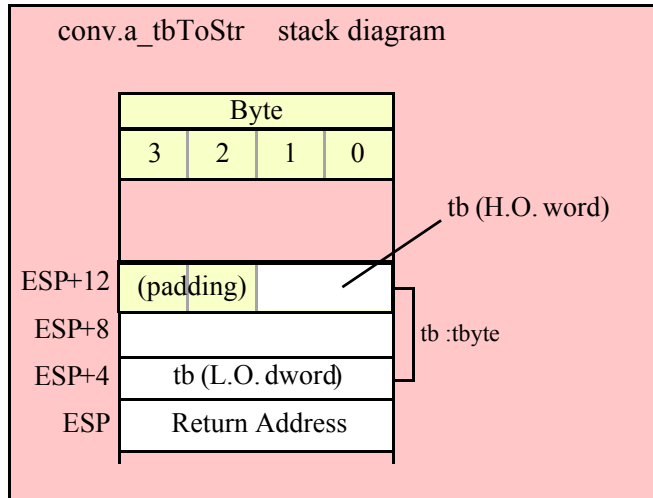
```

push( (type word tbyteVariable[8])); // H.O. word first
push( (type dword tbyteVariable[4]));
push( (type dword qwordVariable[0])); // L.O. dword last
call conv.a_tbToStr;
mov( eax, destStr );

// Passing a constant:

pushd( <constant> >> 64 ); // Push H.O. word as dword, first
pushd( (<constant> >> 32) & $FFFF_FFFF );
pushd( <constant> & $FFFF_FFFF ); // Push L.O. dword last.
call conv.a_tbToStr;
mov( eax, destStr );

```



**conv.lToStr( l:lword; dest:string );**

Converts the 128-bit value of the *l* parameter to 32- or 19-byte string (based on the setting of the underscores flag) that is the hexadecimal representation of this value. If the internal underscores flag is true, then this function emits an underscore between each group of four hexadecimal digits (resulting in a 39-character string). This function raises a string overflow error if the destination string is not large enough to hold the converted string.

HLA high-level calling sequence examples:

```

// The following will push the value of "lwordVariable"
// and destStr onto the stack, and then call conv.lToStr:

conv.lToStr( lwordVariable, destStr );

// The following pushes the constant onto the stack and calls
// conv.lToStr:

conv.lToStr( <constant>, edx ); // EDX contains string pointer value.

```

HLA low-level calling sequence examples:

```

// Passing an lword variable:

push( (type dword lwordVariable[12])); // H.O. dword first
push( (type dword lwordVariable[8]));

```

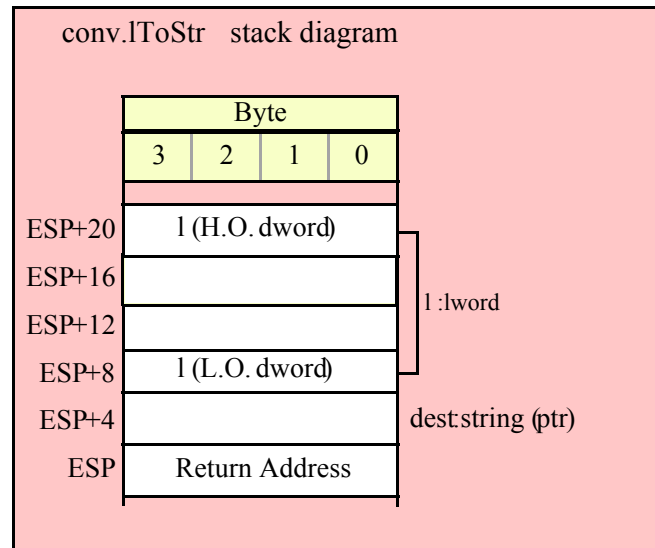
```

    push( (type dword lwordVariable[4]));
    push( (type dword lwordVariable[0])); // L.O. dword last
    push( destStr );
    call conv.lToStr;

// Passing a constant:

    pushd( <constant> >> 96 ); // Push H.O. dword of constant first.
    pushd( (<constant> >> 64) & $FFFF_FFFF );
    pushd( (<constant> >> 32) & $FFFF_FFFF );
    pushd( <constant> & $FFFF_FFFF ); // Push L.O. dword last.
    push( edx ); // EDX contains string pointer value.
    call conv.lToStr;

```



```
conv.a_lToStr( l:lword; dest:string ); @returns( "eax" );
```

Converts the 128-bit value of the *l* parameter to 32- or 19-byte string (based on the setting of the underscores flag) that is the hexadecimal representation of this value. If the internal underscores flag is true, then this function emits an underscore between each group of four hexadecimal digits (resulting in a 39-character string). This function allocates storage for the string on the heap and returns a pointer to that string in the EAX register.

HLA high-level calling sequence examples:

```

// The following will push the value of "lwordVariable"
// onto the stack, and then call conv.a_lToStr:

conv.a_lToStr( lwordVariable );
mov( eax, destStr );

// The following pushes the constant onto the stack and calls
// conv.a_lToStr:

conv.a_lToStr( <constant> );
mov( eax, destStr );

```

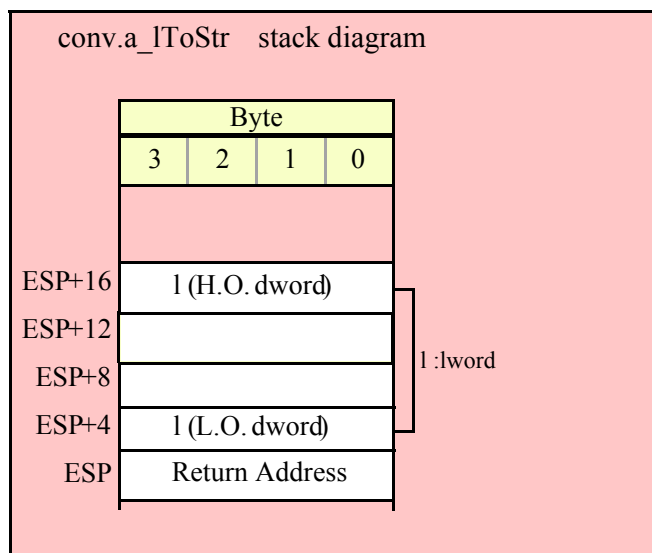
HLA low-level calling sequence examples:

```
// Passing an lword variable:
```

```
push( (type dword lwordVariable[12])); // H.O. dword first
push( (type dword lwordVariable[8]));
push( (type dword lwordVariable[4]));
push( (type dword lwordVariable[0])); // L.O. dword last
call conv.a_lToStr;
mov( eax, destStr );
```

```
// Passing a constant:
```

```
pushd( <constant> >> 96 ); // Push H.O. dword of constant first.
pushd( (<constant> >> 64) & $FFFF_FFFF );
pushd( (<constant> >> 32) & $FFFF_FFFF );
pushd( <constant> & $FFFF_FFFF ); // Push L.O. dword last.
call conv.a_lToStr;
mov( eax, destStr );
```



### 8.3.4.2 Variable-Length Numeric to Hexadecimal String Conversions

The functions in this category convert a numeric value (8, 16, 32, 64, 80, or 128 bits) to a variable-length hexadecimal string (with no leading zeros). The string will contain the minimum number of digits needed to represent the value (plus underscores between each group of four digits if the internal underscores flag contains true).

```
procedure conv.h8ToStr ( b:byte; width:int32; fill:char; buffer:string );
```

Converts the 8-bit value of the b parameter to a one or two-byte string that is the hexadecimal representation of this value. Because 8-bit hexadecimal conversions require no more than two digits, the internal underscores flag setting does not affect the operation of this function. This function raises a string overflow error if the destination string is not large enough to hold the converted string.

HLA high-level calling sequence examples:

```
// The following will push "byteVariable" and "destStr" (which
// is a pointer to a string object) and then call conv.h8ToStr:

conv.h8ToStr( byteVariable, destStr );
```

```
// The following call will BH's value onto the stack and then
// push EDX's value (which is the address of a string object)
// before calling conv.h8ToStr:

conv.h8ToStr( bh, edx );

// The following pushes the constant and destStr and calls
// conv.h8ToStr:

conv.h8ToBuf( <constant>, destStr ); // <constant> must fit in 8 bits
```

When using the HLA high-level calling form, remember that string variables are dword pointers that contain the address of a string object. The destination string parameter is passed by value, not by reference; it just turns out that the value of a string is a pointer to the actual string data (confusing, isn't it?). In any case, this is why you can pass a register value as the destination string location rather than having to pass something like "[edx]". A construct like "[edx]" would imply that EDX contains the address of the string variable, that is, a pointer to the pointer to the string.

HLA low-level calling sequence examples:

```
// Passing a byte variable and a buffer variable, option 1
// (if a 32-bit register is available):

movzx( byteVariable, eax );
push( eax );
push( destStr );
call conv.h8ToStr;

// Passing a byte variable and a buffer variable, option 2
// (if byteVariable isn't the last byte in mapped memory):

push( (type dword byteVariable));
push( destStr );
call conv.h8ToStr;

// Passing a byte variable and a buffer variable, option 3
// No registers are available and we can't guarantee that
// the three bytes following byteVariable are present in
// mapped memory:

sub( 4, esp );
push( eax );
movzx( byteVariable, eax );
mov( eax, [esp+4] );
pop( eax );
push( destStr );
call conv.h8ToStr;

// Passing a pair of registers (hex value in AL, BL, CL, or DL):
// BL = value to print, EDX = pointer to string object.

push( ebx );          // Pushes BL
push( edx );
call conv.h8ToStr;

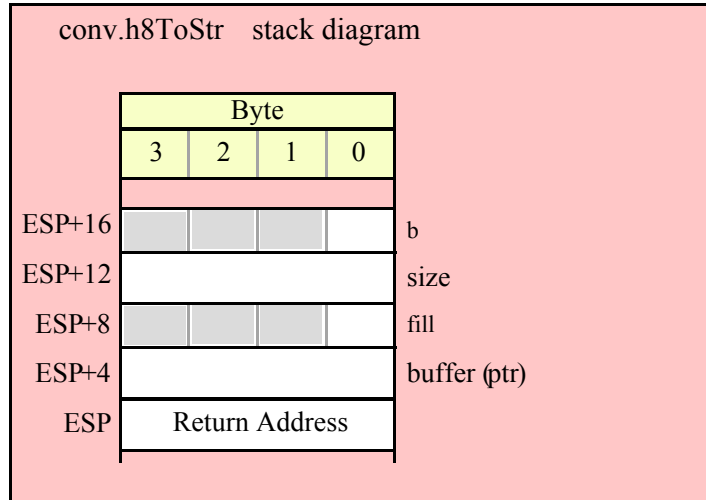
// Passing a pair of registers (hex value in AH, BH, CH, or DH):
// BH = value to print, EDX = pointer to string object.

pushd( 0 );
```

```
mov( bh, [esp] );
push( edx );
call conv.h8ToStr;

// Passing a constant:

pushd( <constant> );
push( destStr );
call conv.h8ToStr;
```



```
procedure conv.a_h8ToStr( b:byte; width:int32; fill:char );
    @returns( "eax" );
```

Allocates an appropriate amount of string storage on the heap and then converts the 8-bit value of the `b` parameter to a 1..2 byte string that is the hexadecimal representation of this value. Returns a pointer to the newly allocated string in EAX. Because the number of digits is always two or less, the internal underscores flag does not affect the string this function produces.

HLA high-level calling sequence examples:

```
// The following will push "byteVariable" and then call conv.a h8ToStr:
```

```
conv.a_h8ToStr( byteVariable );  
mov( eax, byteStr );
```

```
// The following call will BH's value onto the stack
// before calling conv.a h8ToStr:
```

```
conv.a_h8ToStr( bh );  
mov( eax, byteStr );
```

```
// The following pushes the constant and calls
// conv.a h8ToStr:
```

```
conv.a_h8ToStr( <constant> ); // <constant> must fit in 8 bits
mov( eax, byteStr );
```

HLA low-level calling sequence examples:

```
// Passing a byte variable and a buffer variable, option 1
// (if a 32-bit register is available):
```



```

movzx( byteVariable, eax );
push( eax );
call conv.a_h8ToStr;
mov( eax, destStr );

// Passing a byte variable and a buffer variable, option 2
// (if byteVariable isn't the last byte in mapped memory):

push( (type dword byteVariable));
call conv.a_h8ToStr;
mov( eax, destStr );

// Passing a byte variable, option 3
// No registers are available and we can't guarantee that
// the three bytes following byteVariable are present in
// mapped memory:

sub( 4, esp );
push( eax );
movzx( byteVariable, eax );
mov( eax, [esp+4] );
pop( eax );
call conv.a_h8ToStr;
mov( eax, destStr );

// Passing a pair of registers (hex value in AL, BL, CL, or DL):
// BL = value to print, EDX = pointer to string object.

push( ebx );          // Pushes BL
call conv.a_h8ToStr;
mov( eax, byteStr );

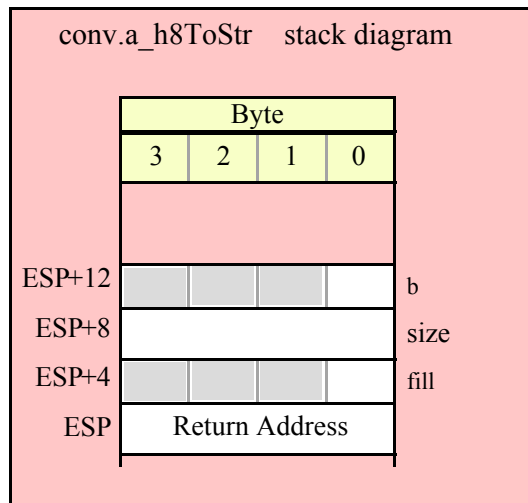
// Passing a pair of registers (hex value in AH, BH, CH, or DH):
// BH = value to print, EDX = pointer to string object.

pushd( 0 );
mov( bh, [esp] );
call conv.a_h8ToStr;
mov( eax, byteStr );

// Passing a constant:

pushd( <constant> );
call conv.a_h8ToStr;
mov( eax, byteStr );

```



```
procedure conv.h16ToStr ( w:word; width:int32; fill:char; buffer:string );
```

Converts the 16-bit value of the w parameter to a 1..4 byte string that is the hexadecimal representation of this value. Because 16-bit hexadecimal conversions require no more than four digits, the internal underscores flag setting does not affect the operation of this function. This function raises a string overflow error if the destination string is not large enough to hold the converted string.

HLA high-level calling sequence examples:

```
// The following will push "wordVariable" and "destStr" (which
// is a pointer to a string object) and then call conv.h16ToStr:
```

```
conv.h16ToStr( wordVariable, destStr );
```

```
// The following call will BX's value onto the stack and then
// push EDX's value (which is the address of a string object)
// before calling conv.h16ToStr:
```

```
conv.h16ToStr( bx, edx );
```

```
// The following pushes the constant and destStr and calls
// conv.h16ToStr:
```

```
conv.h16ToStr( <constant>, destStr ); // <constant> must fit in 16 bits
```

When using the HLA high-level calling form, remember that string variables are dword pointers that contain the address of a string object. The destination string parameter is passed by value, not by reference; it just turns out that the value of a string is a pointer to the actual string data (confusing, isn't it?). In any case, this is why you can pass a register value as the destination string location rather than having to pass something like "[edx]". A construct like "[edx]" would imply that EDX contains the address of the string variable, that is, a pointer to the pointer to the string.

HLA low-level calling sequence examples:

```
// Passing a word variable and a buffer variable, option 1
// (if a 32-bit register is available):
```

```
movzx( byteVariable, eax );
```

```
push( eax );
push( destStr );
call conv.h16ToStr;

// Passing a word variable and a buffer variable, option 2
// (if wordVariable isn't the last byte in mapped memory):

push( (type dword wordVariable));
push( destStr );
call conv.h16ToStr;

// Passing a word variable and a buffer variable, option 3
// No registers are available and we can't guarantee that
// the two bytes following wordVariable are present in
// mapped memory:

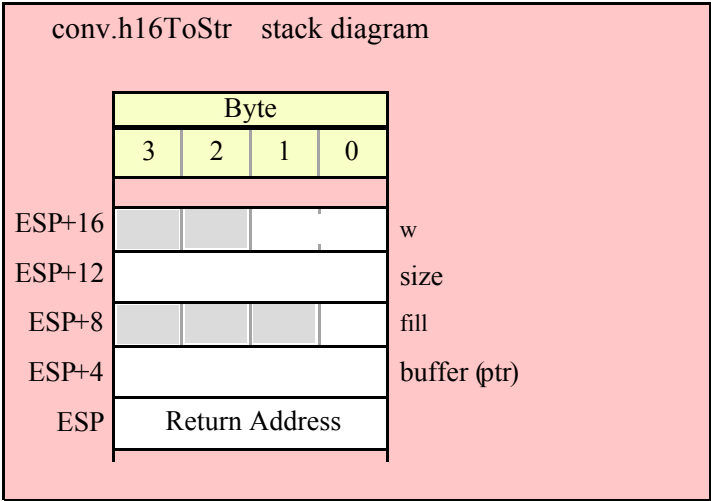
pushw( 0 );
push( wordVariable );
push( destStr );
call conv.h16ToStr;

// Passing a pair of registers:
// BX = value to print, EDX = pointer to string object.

push( ebx );      // Pushes BX
push( edx );
call conv.h16ToStr;

// Passing a constant:

pushd( <constant> );
push( destStr );
call conv.h16ToStr;
```



```

procedure conv.a_h16ToStr( w:word; width:int32; fill:char );
    @returns( "eax" );

```

Allocates an appropriate amount of string storage on the heap and then converts the 16-bit value of the w parameter to a 1..4 byte string that is the hexadecimal representation of this value. Returns a pointer to the newly allocated string in EAX. Because the number of digits is always four or less, the internal underscores flag does not affect the string this function produces.

HLA high-level calling sequence examples:

```

// The following will push "wordVariable" and call conv.a_h16ToStr:

conv.a_h16ToStr( wordVariable );
mov( eax, destStr );

// The following call will BX's value onto the stack
// before calling conv.a_h16ToStr:

conv.a_h16ToStr( bx );

// The following pushes the constant and calls
// conv.a_h16ToStr:

conv.a_h16ToStr( <const>, destStr ); // <const> must fit in 16 bits

```

HLA low-level calling sequence examples:

```

// Passing a word variable and a buffer variable, option 1
// (if a 32-bit register is available):

movzx( wordVariable, eax );
push( eax );
call conv.a_h16ToStr;
mov( eax, destStr );

// Passing a word variable and a buffer variable, option 2
// (if wordVariable isn't the last byte in mapped memory):

push( (type dword wordVariable) );
call conv.a_h16ToStr;
mov( eax, destStr );

// Passing a word variable and a buffer variable, option 3
// No registers are available and we can't guarantee that
// the two bytes following wordVariable are present in
// mapped memory:

pushw( 0 );
push( wordVariable );
call conv.a_h16ToStr;
mov( eax, destStr );

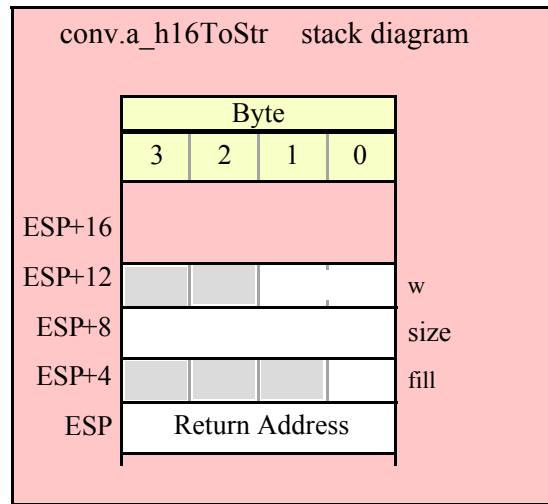
// Passing a pair of registers:
// BX = value to print.

push( ebx );          // Pushes BX
call conv.a_h16ToStr;
mov( eax, wordStr );

```

```
// Passing a constant:

pushd( <constant> );
call conv.a_h16ToStr;
mov( eax, destStr );
```



```
procedure conv.h32ToStr ( d:dword; width:int32; fill:char; buffer:string );
```

Converts the 32-bit value of the *d* parameter to a 1..8 byte string (1..9 bytes if the underscores flag is set) that is the hexadecimal representation of this value. If the 32-bit value is greater than \$FFFF and the internal underscores flag is true, then this function will insert an underscore between the 4<sup>th</sup> and 5<sup>th</sup> digits of the string it produces. This function raises a string overflow error if the destination string is not large enough to hold the converted string.

HLA high-level calling sequence examples:

```
// The following will push "dwordVariable" and "destStr" (which
// is a pointer to a string object) and then call conv.h32ToStr:

conv.h32ToStr( dwordVariable, destStr );

// The following call will push EBX's value onto the stack and then
// push EDX's value (which is the address of a string object)
// before calling conv.h32ToStr:

conv.h32ToStr( ebx, edx );

// The following pushes the constant and destStr and calls
// conv.h32ToStr:

conv.h32ToStr( <constant>, destStr ); // <constant> must fit in 32 bits
```

When using the HLA high-level calling form, remember that string variables are dword pointers that contain the address of a string object. The destination string parameter is passed by value, not by reference; it just turns out that the value of a string is a pointer to the actual string data (confusing, isn't it?). In any case, this is why you can pass a register value as the destination string location rather than having to pass something like "[edx]". A construct like "[edx]" would imply that EDX contains the address of the string variable, that is, a pointer to the pointer to the string.

HLA low-level calling sequence examples:

```
// Passing a dword variable and a buffer variable:

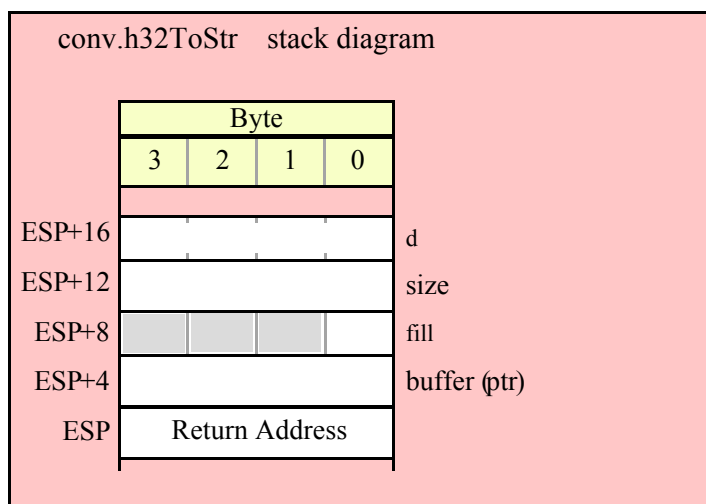
push( dwordVariable );
push( destStr );
call conv.h32ToStr;

// Passing a pair of registers:
// EBX = value to print, EDX = pointer to string object.

push( ebx );
push( edx );
call conv.h32ToStr;

// Passing a constant:

pushd( <constant> );
push( destStr );
call conv.h32ToStr;
```



```
procedure conv.a_h32ToStr( d:dword; width:int32; fill:char );
    @returns( "eax" );
```

Allocates an appropriate amount of string storage on the heap and then converts the 32-bit value of the `d` parameter to a 1..8 byte string (1..9 bytes if the underscores flag is set) that is the hexadecimal representation of this value. Returns a pointer to the newly allocated string in EAX. If the 32-bit value is greater than \$FFFF and the internal underscores flag is true, then this function will insert underscores between each set of four digits to the left of the least significant digit in the string it produces.

HLA high-level calling sequence examples:

```
// The following will push "dwordVariable" and then call conv.a_h32ToStr:

conv.a_h32ToStr( dwordVariable );
mov( eax, destStr );

// The following call will push EBX's value onto the stack
```

```
// before calling conv.a_h32ToStr:
conv.a_h32ToStr( ebx );

// The following pushes the constant and calls
// conv.a_h32ToStr:
conv.a_h32ToStr( <constant>, destStr ); // <constant> must fit in 32 bits
```

HLA low-level calling sequence examples:

```
// Passing a dword variable:
```

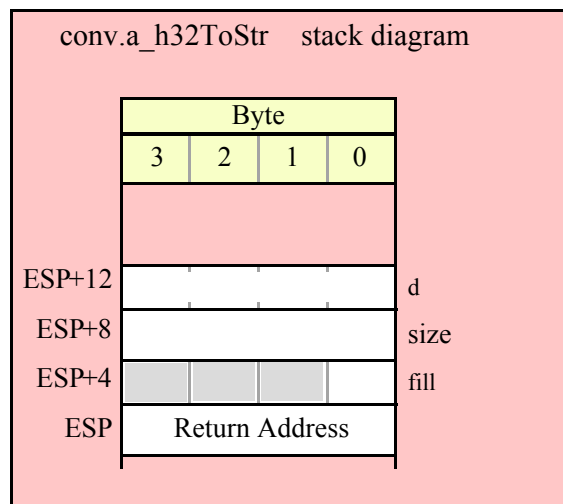
```
push( dwordVariable );
call conv.a_h32ToStr;
mov( eax, destStr );
```

```
// Passing a register:  
// EBX = value to print.
```

```
push( ebx );
call conv.a_h32ToStr;
mov( eax, dwordStr );
```

```
// Passing a constant:
```

```
pushd( <constant> );  
call conv.a_h32ToStr;  
mov( eax, destStr );
```



```
procedure conv.h64ToStr ( q:qword; width:int32; fill:char; buffer:string );
```

Converts the 64-bit value of the q parameter to a 1..16 byte string (1..19 bytes if the underscores flag is set) that is the hexadecimal representation of this value. If the 64-bit value is greater than \$FFFF and the internal underscores flag is true, then this function will insert underscores between each set of four digits to the left of the least significant digit in the string it produces. This function raises a string overflow error if the destination string is not large enough to hold the converted string.

HLA high-level calling sequence examples:

```
// The following will push the value of "qwordVariable"
// and the value of the destStr string variable
// onto the stack and then call conv.h64ToStr:
```

```
conv.h64ToStr( qwordVariable, destStr );
```

```
// The following pushes the constant onto the stack along with
// the value held in the destStr variable and calls
// conv.h64ToStr:
```

```
conv.h64ToStr( <constant>, destStr ); // <constant> must fit in 64 bits
```

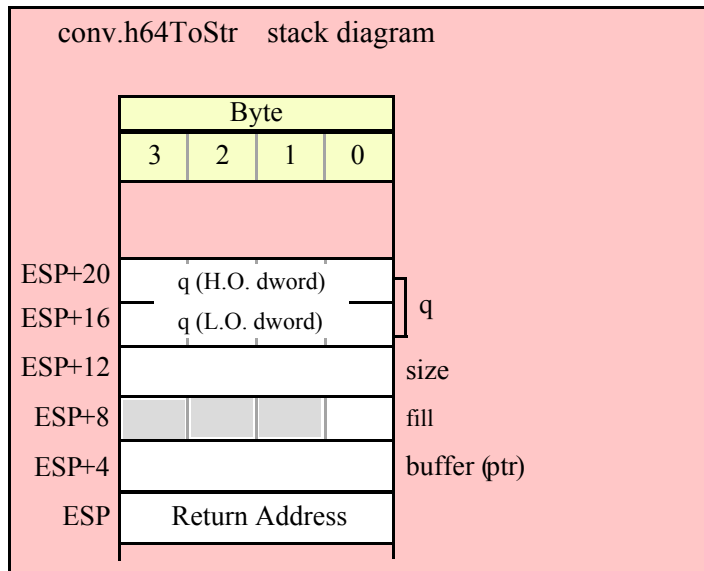
HLA low-level calling sequence examples:

```
// Passing a qword variable and a buffer variable:
```

```
    push( (type dword qwordVariable[4]));// H.O. dword first
    push( (type dword qwordVariable[0]));// L.O. dword last
push( destStr );
call conv.h64ToStr;
```

```
// Passing a constant:
```

```
pushd( <constant> >> 32 );// Push H.O. dword of constant first.
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword second.
push( destStr );
call conv.h64ToStr;
```





```

procedure conv.a_h64ToStr( q:qword;  width:int32; fill:char );
  @returns( "eax" );

```

Allocates an appropriate amount of string storage on the heap and then converts the 64-bit value of the q parameter to a 1..16 byte string (1..19 bytes if the underscores flag is set) that is the hexadecimal representation of this value. Returns a pointer to the newly allocated string in EAX. If the 64-bit value is greater than \$FFFF and the internal underscores flag is true, then this function will insert underscores between each set of four digits to the left of the least significant digit in the string it produces.

HLA high-level calling sequence examples:

```

// The following will push the value of "qwordVariable"
// onto the stack and then call conv.a_h64ToStr:

```

```

conv.a_h64ToStr( qwordVariable );
mov( eax, destStr );

```

```

// The following pushes the constant onto the stack and calls
// conv.a_h64ToStr:

```

```

conv.a_h64ToStr( <constant> ); // <constant> must fit in 64 bits
mov( eax, destStr );

```

HLA low-level calling sequence examples:

```

// Passing a qword variable:

```

```

                push( (type dword qwordVariable[4])); // H.O. dword first
                push( (type dword qwordVariable[0])); // L.O. dword last
call conv.a_h64ToStr;
mov( eax, destStr );

```

```

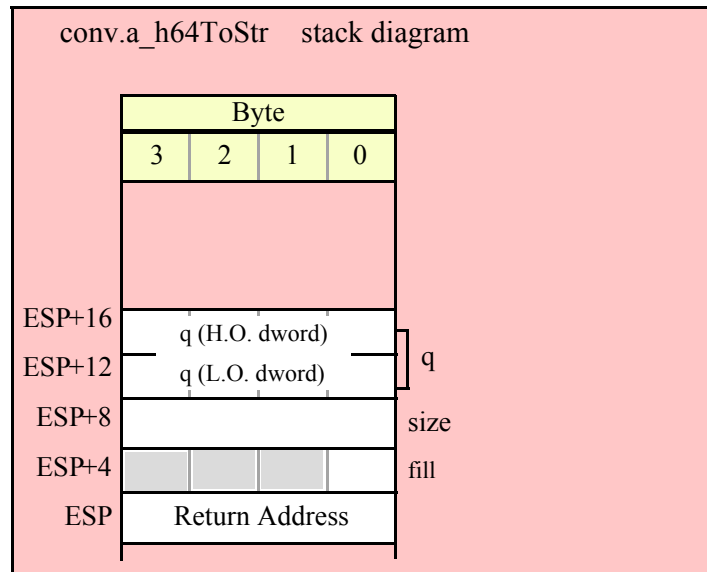
// Passing a constant:

```

```

pushd( <constant> >> 32 ); // Push H.O. dword of constant first.
pushd( <constant> & $FFFF_FFFF ); // Push L.O. dword second.
call conv.a_h64ToStr;
mov( eax, destStr );

```



```
procedure conv.h80ToStr( tb:tbyte; width:int32; fill:char; buffer:string );
```

Converts the 80-bit value of the tb parameter to a 1..20 byte string (1..24 bytes if the underscores flag is set) that is the hexadecimal representation of this value. If the 80-bit value is greater than \$FFFF and the internal underscores flag is true, then this function will insert underscores between each set of four digits to the left of the least significant digit in the string it produces. This function raises a string overflow error if the destination string is not large enough to hold the converted string.

HLA high-level calling sequence examples:

```
// The following will push the value of "tbyteVariable"
// and the value of destStr onto the stack
// and then call conv.h80ToStr:
```

```
conv.h80ToStr( tbyteVariable, destStr );
```

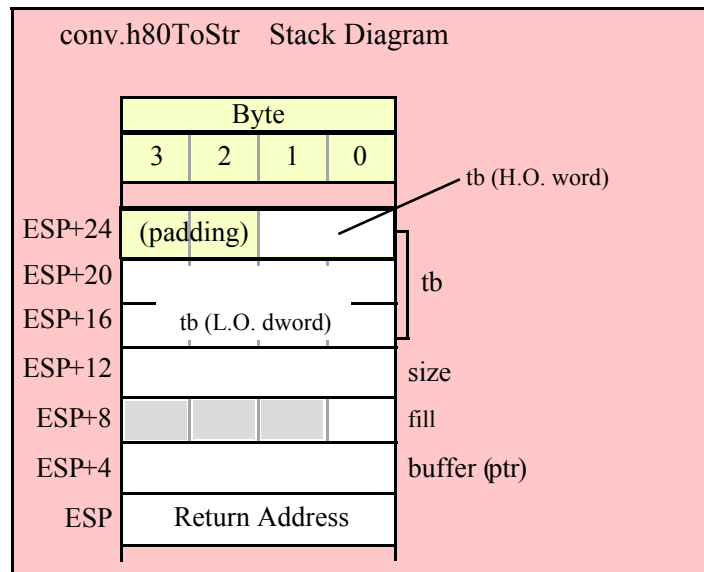
HLA low-level calling sequence examples:

```
// Passing a tbyte variable:
```

```
pushw( 0 ); // Must pad parameter to 12 bytes
push( (type word tbyteVariable[8])); // H.O. word first
push( (type dword tbyteVariable[4]));
push( (type dword qwordVariable[0])); // L.O. dword last
push( destStr );
call conv.h80ToStr;
```

```
// Passing a constant:
```

```
pushd( <constant> >> 64 ); // Push H.O. word as dword, first
pushd( (<constant> >> 32) & $FFFF_FFFF );
pushd( <constant> & $FFFF_FFFF ); // Push L.O. dword last.
push( destStr );
call conv.h80ToStr;
```



```

procedure conv.a_h80ToStr( tb:tbyte; width:int32; fill:char );
    @returns( "eax" );

```

Allocates an appropriate amount of string storage on the heap and then converts the 80-bit value of the `tb` parameter to a 1..20 byte string (1..24 bytes if the underscores flag is set) that is the hexadecimal representation of this value. Returns a pointer to the newly allocated string in EAX. If the 80-bit value is greater than \$FFFF and the internal underscores flag is true, then this function will insert underscores between each set of four digits to the left of the least significant digit in the string it produces.

HLA high-level calling sequence examples:

```

// The following will push the value of "tbyteVariable"
// onto the stack and then call conv.a_h80ToStr:

```

```

conv.a_h80ToStr( tbyteVariable );
mov( eax, destStr );

```

HLA low-level calling sequence examples:

```

// Passing a tbyte variable:

```

```

pushw( 0 ); // Must pad parameter to 12 bytes
push( (type word tbyteVariable[8])); // H.O. word first
push( (type dword tbyteVariable[4]));
push( (type dword qwordVariable[0])); // L.O. dword last
call conv.a_h80ToStr;
mov( eax, destStr );

```

```

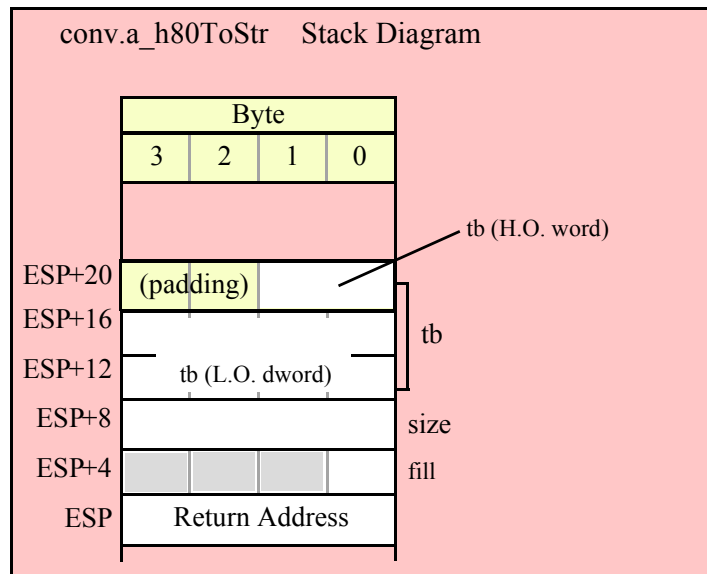
// Passing a constant:

```

```

pushd( <constant> >> 64 ); // Push H.O. word as dword, first
pushd( (<constant> >> 32) & $FFFF_FFFF );
pushd( <constant> & $FFFF_FFFF ); // Push L.O. dword last.
call conv.a_h80ToStr;
mov( eax, destStr );

```



```
procedure conv.h128ToStr ( l:lword; width:int32; fill:char; buffer:string );
```

Converts the 128-bit value of the *l* parameter to a 1..32 byte string (1..39 bytes if the underscores flag is set) that is the hexadecimal representation of this value. If the 128-bit value is greater than \$FFFF and the internal underscores flag is true, then this function will insert underscores between each set of four digits to the left of the least significant digit in the string it produces. This function raises a string overflow error if the destination string is not large enough to hold the converted string.

HLA high-level calling sequence examples:

```
// The following will push the value of "lwordVariable"
// and destStr onto the stack, and then call conv.h128ToStr:
```

```
conv.h128ToStr( lwordVariable, destStr );
```

```
// The following pushes the constant onto the stack and calls
// conv.h128ToStr:
```

```
conv.h128ToStr( <constant>, edx ); // EDX contains string pointer value.
```

HLA low-level calling sequence examples:

```
// Passing an lword variable:
```

```
push( (type dword lwordVariable[12])); // H.O. dword first
push( (type dword lwordVariable[8]));
push( (type dword lwordVariable[4]));
push( (type dword lwordVariable[0])); // L.O. dword last
push( destStr );
call conv.h128ToStr;
```

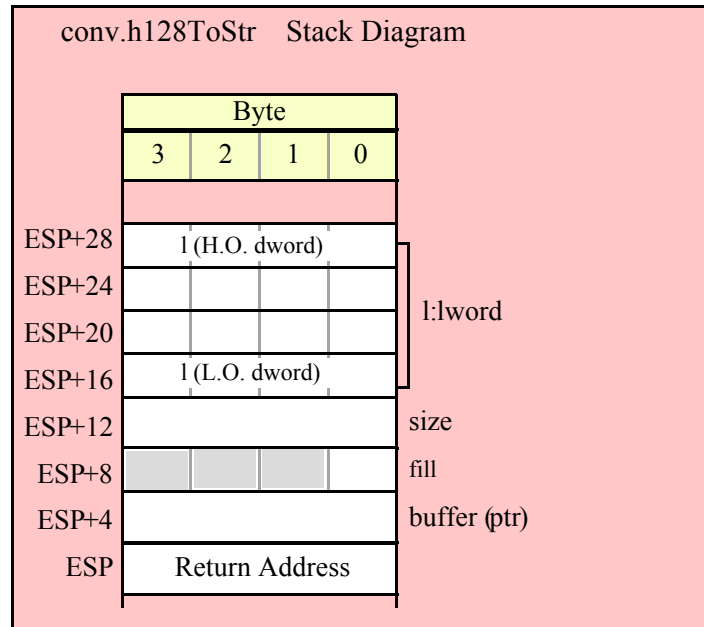
```
// Passing a constant:
```

```
pushd( <constant> >> 96 ); // Push H.O. dword of constant first.
```

```

pushd( (<constant> >> 64)& $FFFF_FFFF );
pushd( (<constant> >> 32)& $FFFF_FFFF );
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword last.
push( edx );// EDX contains string pointer value.
call conv.h128ToStr;

```



```

conv.a_h128ToStr( l:ldword; width:int32; fill:char );
@returns( "eax" );

```

Allocates an appropriate amount of string storage on the heap and then converts the 128-bit value of the `l` parameter to a 1..32 byte string (1..39 bytes if the underscores flag is set) that is the hexadecimal representation of this value. Returns a pointer to the newly allocated string in EAX. If the 128-bit value is greater than \$FFFF and the internal underscores flag is true, then this function will insert underscores between each set of four digits to the left of the least significant digit in the string it produces.

HLA high-level calling sequence examples:

```

// The following will push the value of "lwordVariable"
// onto the stack, and then call conv.a_h128ToStr:

conv.a_h128ToStr( lwordVariable );
mov( eax, destStr );

// The following pushes the constant onto the stack and calls
// conv.a_h128ToStr:

conv.a_h128ToStr( <constant> );
mov( eax, destStr );

```

HLA low-level calling sequence examples:

```

// Passing an lword variable:

push( (type dword lwordVariable[12]));// H.O. dword first
push( (type dword lwordVariable[8]));
push( (type dword lwordVariable[4]));
push( (type dword lwordVariable[0]));// L.O. dword last

```

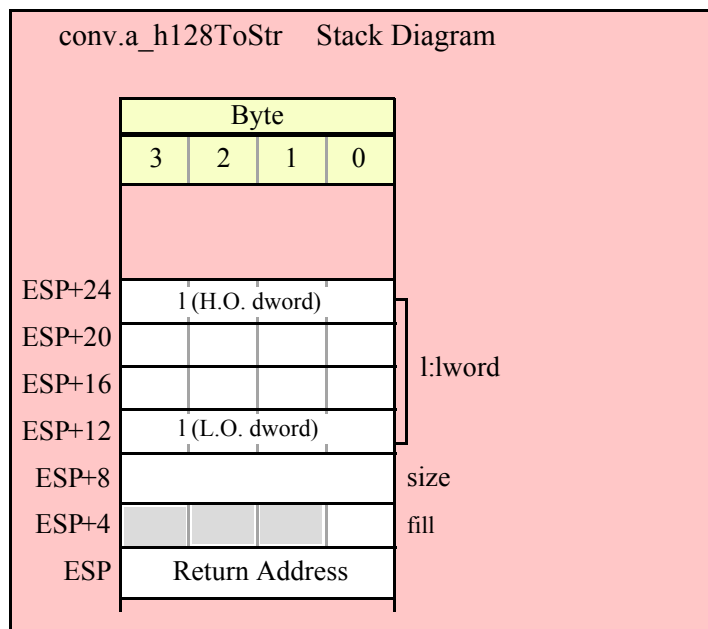
```

call conv.a_h128ToStr;
mov( eax, destStr );

// Passing a constant:

pushd( <constant> >> 96 );// Push H.O. dword of constant first.
pushd( (<constant> >> 64)& $FFFF_FFFF );
pushd( (<constant> >> 32)& $FFFF_FFFF );
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword last.
call conv.a_h128ToStr;
mov( eax, destStr );

```



### 8.3.5 Hexadecimal Buffer to Numeric Conversions

The hexadecimal buffer to numeric conversions ("ASCII to hexadecimal") convert a sequence of characters at some address in memory to numeric form.

These routines will skip over any leading underscore and delimiter characters (specified by the internal delimiters character set, see the discussion of `conv.setDelimiters` and `conv.getDelimiters` for details). These functions will convert all characters in the sequence until encountering a non-hexadecimal digit or underscore. If the first non-hex character is not the end of string or a delimiter character, these functions will raise a conversion exception. If the character is not a valid 7-bit ASCII character, these functions will raise an illegal character exception.

```

procedure conv.atoh8( var buffer:var in esi ); @returns( "eax" );

```

This function converts the hexadecimal character sequence beginning at character position held in ESI to an 8-bit numeric value. It raises an overflow exception if the value is outside the range \$00..\$FF. This function returns the result in AL, zero extended into EAX. This function returns ESI pointing at the first character after the sequence of hexadecimal characters.

HLA high-level calling sequence examples:

```

// The following will convert the hexadecimal characters in memory
// at the address specified by [esi] into a numeric value
// and return that value in AL:

```

```

conv.atoh8( [esi] );
mov( al, hexNumericResult );

// The following loads ESI with the address of
// a sequence of hexadecimal characters (held in an HLA
// string) and converts them to an 8-bit number:

conv.atoh8( sourceStr ); // Loads "sourceStr" into ESI
mov( al, byteVariable );

```

HLA low-level calling sequence examples:

```

// Same as first example above - ESI already contains
// the address of the first character to convert:

call conv.atoh8;
mov( al, hexNumericResult );

// Same as second example above

static
  sourceStr :string := "12";
  .
  .
  .
mov( sourceStr, esi );
call conv.atoh8;
mov( al, hex12 );

```

**procedure conv.atoh16( var buffer:var in esi ); @returns( "eax" );**

This function converts the hexadecimal character sequence beginning at character position held in ESI to an 8-bit numeric value. It raises an overflow exception if the value is outside the range \$0000..\$FFFF. This function returns the result in AX, zero extended into EAX. This function returns ESI pointing at the first character after the sequence of hexadecimal characters.

HLA high-level calling sequence examples:

```

// The following will convert the hexadecimal characters in memory
// at the address specified by [esi] into a numeric value
// and return that value in AX:

conv.atoh16( [esi] );
mov( ax, hex16NumericResult );

// The following loads ESI with the address of
// a sequence of hexadecimal characters (held in an HLA
// string) and converts them to a 16-bit number:

conv.atoh16( sourceStr ); // Loads "sourceStr" into ESI
mov( ax, wordVariable );

```

HLA low-level calling sequence examples:

```
// Same as first example above - ESI already contains
// the address of the first character to convert:
```

```
    call conv.atoh16;
mov( ax, hex16NumericResult );
```

```
// Same as second example above
```

```
static
    sourceStr :string := "12";
.
.
.
mov( sourceStr, esi );
    call conv.atoh16;
    mov( ax, hex12 );
```

**procedure conv.atoh32( var buffer:var in esi ); @returns( "eax" );**

This function converts the hexadecimal character sequence beginning at character position held in ESI to an 8-bit numeric value. It raises an overflow exception if the value is outside the range \$0000\_0000..\$FFFF\_FFFF. This function returns the result in EAX. This function returns ESI pointing at the first character after the sequence of hexadecimal characters.

HLA high-level calling sequence examples:

```
// The following will convert the hexadecimal characters in memory
// at the address specified by [esi] into a numeric value
// and return that value in EAX:
```

```
conv.atoh32( [esi] );
mov( eax, hex32NumericResult );
```

```
// The following loads ESI with the address of
// a sequence of hexadecimal characters (held in an HLA
// string) and converts them to a 32-bit number:
```

```
conv.atoh32( sourceStr ); // Loads "sourceStr" into ESI
mov( eax, dwordVariable );
```

HLA low-level calling sequence examples:

```
// Same as first example above - ESI already contains
// the address of the first character to convert:
```

```
    call conv.atoh32;
mov( eax, hex32NumericResult );
```

```
// Same as second example above
```

```
static
    sourceStr :string := "12";
.
.
.
```



```

    .
mov( sourceStr, esi );
    call conv.atoh32;
    mov( eax, hex12 );

```

**procedure conv.atoh64( var buffer:var in esi ); @returns( "edx:eax" );**

This function converts the hexadecimal character sequence beginning at character position held in ESI to an 8-bit numeric value. It raises an overflow exception if the value is outside the range \$0000\_0000\_0000\_0000..\$FFFF\_FFFF\_FFFF\_FFFF. This function returns the result in EDX:EAX (H.O. double word in EDX). This function returns ESI pointing at the first character after the sequence of hexadecimal characters.

HLA high-level calling sequence examples:

```

// The following will convert the hexadecimal characters in memory
// at the address specified by [esi] into a numeric value
// and return that value in EDX:EAX:

```

```

conv.atoh64( [esi] );
mov( eax, (type dword hex64NumericResult[0]) );
mov( edx, (type dword hex64NumericResult[4]) );

```

```

// The following loads ESI with the address of
// a sequence of hexadecimal characters (held in an HLA
// string) and converts them to a 64-bit number:

```

```

conv.atoh64( sourceStr ); // Loads "sourceStr" into ESI
mov( eax, (type dword qwordVariable[0]) );
mov( edx, (type dword qwordVariable[4]) );

```

HLA low-level calling sequence examples:

```

// Same as first example above - ESI already contains
// the address of the first character to convert:

```

```

    call conv.atoh64;
mov( eax, (type dword hex64NumericResult[0]) );
mov( edx, (type dword hex64NumericResult[4]) );

```

```

// Same as second example above

```

```

static
    sourceStr:string := "12";
    .
    .
    .
mov( sourceStr, esi );
    call conv.atoh64;
mov( eax, (type dword qwordVariable[0]) );
mov( edx, (type dword qwordVariable[4]) );

```

```
procedure conv.atoh128( var buffer:var in esi; var dest:lword );
```

This function converts the hexadecimal character sequence beginning at character position in the string to a 128-bit numeric value. It raises an overflow exception if the value is outside the range \$0000\_0000\_0000\_0000\_0000\_0000\_0000\_0000..\$FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF. This function returns the result in the variable specified by the dest parameter.

HLA high-level calling sequence examples:

```
// The following will convert the hexadecimal characters in memory
// at the address specified by [esi] into a numeric value
// and stores that value in lwordDest (passed by reference):
```

```
conv.atoh128( [esi], lwordDest );
```

```
// The following loads ESI with the address of
// a sequence of hexadecimal characters (held in an HLA
// string) and converts them to a 128-bit number that it
// stores in lwordDest:
```

```
conv.atoh128( sourceStr, lwordDest ); // Loads "sourceStr" into ESI
```

HLA low-level calling sequence examples:

```
// Option 1: lwordDest is a static object declared in a
// HLA STATIC, READONLY, or STORAGE section:
// As with the first example above, assume ESI already
// contains the address of the string to convert:
```

```
pushd( &lwordDest );// Pass address of lwordDest as reference parm.
call conv.atoh128;
```

```
// Option 2: lwordDest is a simple automatic variable (no indexing)
// declared in a VAR section (or as a parameter). Assume that
// no 32-bit registers can be disturbed by this code.
// As with the first example above, assume ESI already
// contains the address of the string to convert:
```

```
push( ebp );
add( @offset( lwordDest ), (type dword [esp]));
call conv.atoh128;
```

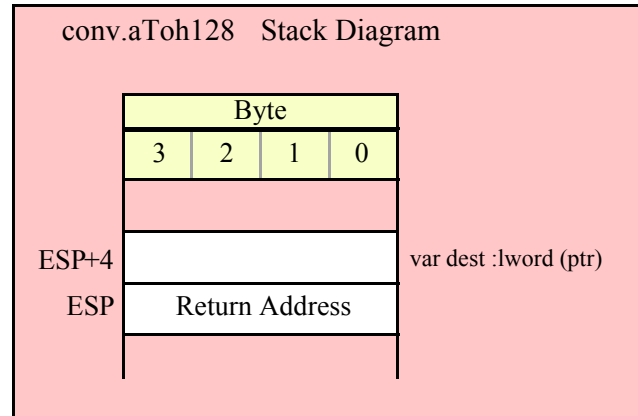
```
// Option 3: lwordDest is a complex addressing mode and at least
// one 32-bit register is available for use by this code.
// As with the first example above, assume ESI already
// contains the address of the string to convert:
```

```
lea( eax, lwordDest );// Assume EAX is the available register
push( eax );
call conv.atoh128;
```

```
// Same as second high-level example above. Assumes that
// lwordDest is a static object.
```

```
static
  sourceStr :string := "12";
.
.
.
mov( sourceStr, esi );
```

```
pushd( &lwordDest );
call conv.atoh128;
```



### 8.3.6 Hexadecimal String to Numeric Conversions

This functions convert a string value, that contain the hexadecimal representation of a number, into the numeric form. These functions have two parameters: a string object and an index into that string. Numeric conversion begins at the zero-based character position specified by the index parameter. For example, the invocation

```
conv.strToh8( someStr, 5 );
```

begins the conversion starting with the sixth character (index 5) in someStr. These functions will raise an "index out of range" exception if the supplied index is greater than the size of the string the first parameter specifies. They will return a null pointer reference exception if the string parameter is NULL (they will return an illegal memory access exception if the first parameter is not a valid pointer and references unpagged memory).

These routines will skip over any leading underscore and delimiter characters (specified by the internal delimiters character set, see the discussion of conv.setDelimiters and conv.getDelimiters for details). These functions will convert all characters in the sequence until encountering a non-hexadecimal digit or underscore. If the first non-hex character is not the end of string or a delimiter character, these functions will raise a conversion exception. If the character is not a valid 7-bit ASCII character, these functions will raise an illegal character exception.

```
procedure conv.strToh8( s:string; index:dword ); @returns( "eax" );
```

This function converts the hexadecimal character sequence beginning at the index<sup>th</sup> character position in the string to an 8-bit numeric value. It raises an overflow exception if the value is outside the range \$00..\$FF. This function returns the result in AL, zero extended into EAX.

HLA high-level calling sequence examples:

```
// The following will convert the characters at the beginning
// of "hexValueStr" to numeric form:
```

```
conv.strToh8( hexValueStr, 0 );// Index=0 starts at beginning
mov( al, hexNumericResult );
```

```
// The following demonstrates using a non-zero index:
```

```
conv.strToh8( "abc12", 3 ); // "12" begins at offset 3
mov( al, hex12 );
```

HLA low-level calling sequence examples:

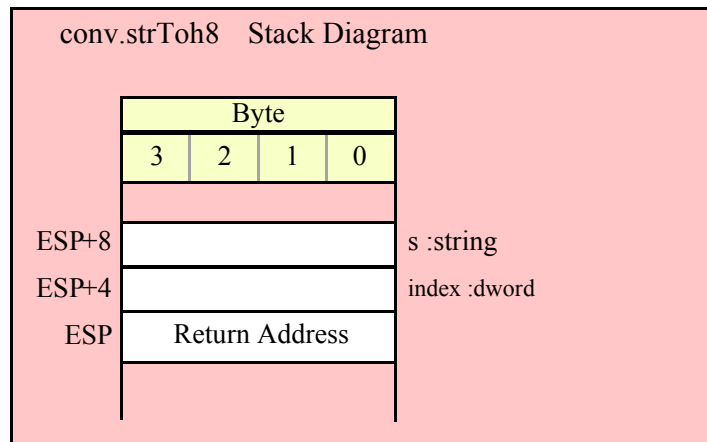
```

    push( hexValueStr );// Same as first example above
    pushd( 0 );
    call conv.strToh8;
    mov( al, hexNumericResult );

```

```
// Same as second example above
```

```
static
    str12 :string := "abc12";
.
.
.
push( str12 );// Note that str12 points at "abc12".
pushd( 3 );// Index to "12" in "abc12".
call conv.strToh8;
mov( al, hex12 );
```



```
procedure conv.strToh16( s:string; index:dword ); @returns( "eax" );
```

This function converts the hexadecimal character sequence beginning at the index<sup>th</sup> character position in the string to a 16-bit numeric value. It raises an overflow exception if the value is outside the range \$0000..\$FFFF. This function returns the result in AX, zero extended into EAX.

HLA high-level calling sequence examples:

```
// The following will convert the characters at the beginning
// of "hexValueStr" to numeric form:
```

```
conv.strToH16( hexValueStr, 0 );// Index=0 starts at beginning
mov( ax, wordVar );
```

```
// The following demonstrates using a non-zero index:
```

```
conv.strToh16( "abc12FF", 3 ); // "12FF" begins at offset 3
mov( ax, wordVar );
```

HLA low-level calling sequence examples:

```

    push( hexValueStr );// Same as first example above
    pushd( 0 );
    call conv.strToh16;
    mov( ax, wordVar );

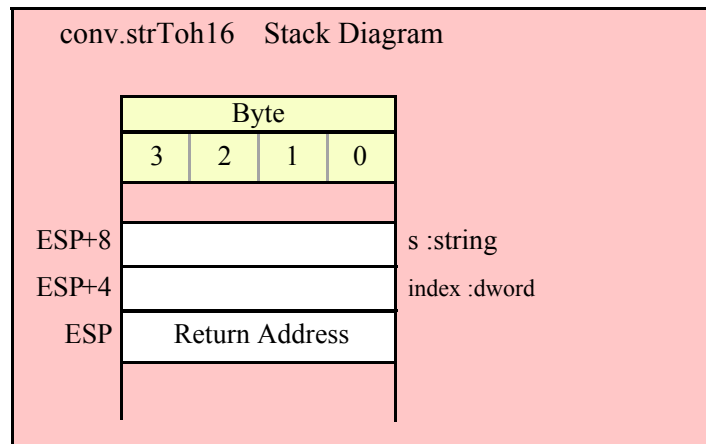
```

// Same as second example above

```

static
    str12FF :string := "abc12FF";
    .
    .
    .
push( str12FF );// Note that str12FF points at "abc12FF".
pushd( 3 );// Index to "12FF" in "abc12FF".
call conv.strToh16;
mov( ax, wordVar );

```



```
procedure conv.strToh32( s:string; index:dword ); @returns( "eax" );
```

This function converts the hexadecimal character sequence beginning at the index<sup>th</sup> character position in the string to a 32-bit numeric value. It raises an overflow exception if the value is outside the range \$0000\_0000..\$FFF\_FFFF. This function returns the result in EAX.

HLA high-level calling sequence examples:

```

// The following will convert the characters at the beginning
// of "hexValueStr" to numeric form:

```

```

conv.strToh32( hexValueStr, 0 );// Index=0 starts at beginning
mov( eax, dwordVar );

```

```

// The following demonstrates using a non-zero index:

```

```

conv.strToh32( "abc12_FF00", 3 ); // "12_FF00" begins at offset 3
mov( eax, dwordVar );

```

HLA low-level calling sequence examples:

```

    push( hexValueStr );// Same as first example above

```

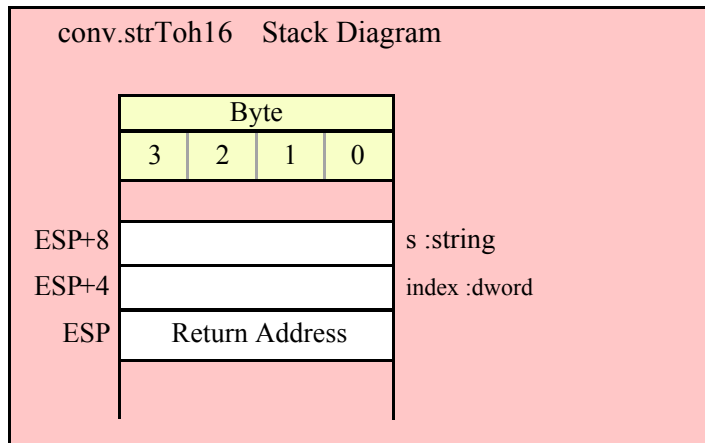
```

pushd( 0 );
call conv.strToh32;
mov( eax, dwordVar );

// Same as second example above

static
    str12FF00 :string := "abc12_FF00";
    .
    .
    .
push( str12FF00 );// Note that str12FF00 points at "abc12_FF00".
pushd( 3 ); // Index to "12_FF00" in "abc12_FF00".
call conv.strToh32;
mov( eax, dwordVar );// dwordVar now contains $12_FF00.

```



```
procedure conv.strToh64( s:string; index:dword ); @returns( "edx:eax" );
```

This function converts the hexadecimal character sequence beginning at the index<sup>th</sup> character position in the string to a 64-bit numeric value. It raises an overflow exception if the value is outside the range \$0000\_0000\_0000\_0000..\$FFFF\_FFFF\_FFFF\_FFFF. This function returns the result in EDX:EAX (H.O. double word in EDX).

HLA high-level calling sequence examples:

```
// The following will convert the characters at the beginning
// of "hexValueStr" to numeric form:
```

```
conv.strToh64( hexValueStr, 0 );// Index=0 starts at beginning
mov( eax, (type dword qwordVar[0]) );
mov( edx, (type dword qwordVar[4]) );
```

```
// The following demonstrates using a non-zero index:
```

```
conv.strToh64( "abc12", 1 ); // "bc12" begins at offset 1
mov( eax, (type dword qwordVar[0]) );
mov( edx, (type dword qwordVar[4]) );
```

HLA low-level calling sequence examples:

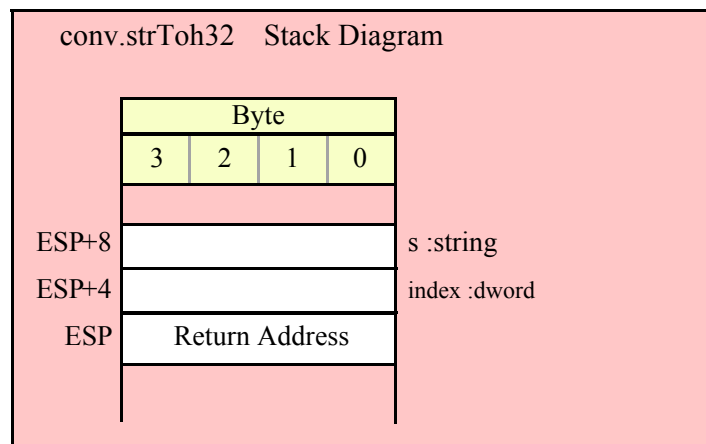
```

    push( hexValueStr ); // Same as first example above
    pushd( 0 );
    call conv.strToh64;
    mov( eax, (type dword qwordVar[0]) );
    mov( edx, (type dword qwordVar[4]) );

    // Same as second example above

    static
        strabc12 :string := "abc12";
        .
        .
        .
    push( strabc12 ); // Note that strabc12 points at "abc12".
    pushd( 1 ); // Index to "bc12" in "abc12".
    call conv.strToh64;
    mov( eax, (type dword qwordVar[0]) );
    mov( edx, (type dword qwordVar[4]) );

```



```
procedure conv.strToh128( s:string; index:dword; var dest:lword );
```

This function converts the hexadecimal character sequence beginning at the index<sup>th</sup> character position in the string to a 128-bit numeric value. It raises an overflow exception if the value is outside the range \$0000\_0000\_0000\_0000\_0000\_0000\_0000\_0000..\$FFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF. This function returns the result in the variable specified by the dest parameter.

HLA high-level calling sequence examples:

```

// The following will convert the characters at the beginning
// of "hexValueStr" (index=0) to numeric form and store the
// 128-bit result into the lwordDest variable:

```

```
conv.strToh128( hexValueStr, 0, lwordDest );
```

```
// The following demonstrates using a non-zero index:
```

```
conv.strToh128( "abc1234567890abcdef", 3, lwordDest );
```

HLA low-level calling sequence examples:

```
// Option #1: lwordDest is a STATIC/READONLY/STORAGE
// variable:

    push( hexValueStr );// Same as first example above
    pushd( 0 );
    pushd( &lwordDest );
    call conv.strToh128;

// Option #2: lwordDest is not a static object and
// a 32-bit register is available for use:

    push( hexValueStr );// Same as first example above
    pushd( 0 );
    lea( eax, lwordDest ); // Assuming EAX is available
    push( eax );
    call conv.strToh128;

// Option #3: lwordDest is an automatic (var) object and
// no 32-bit registers are available for use:

    push( hexValueStr );// Same as first example above
    pushd( 0 );
    push( ebp );
    add( @offset( lwordDest ), (type dword [esp]) );
    call conv.strToh128;

// Option #4: lwordDest is a complex addressing mode object and
// no 32-bit registers are available for use:

    push( hexValueStr );// Same as first example above
    pushd( 0 );
    sub( 4, esp );
    push( eax );
    lea( eax, lwordDest );
    mov( eax, [esp+4] );
    pop( eax );
    call conv.strToh128;
```



## 8.4 Signed Integer Conversions

The integer conversion functions process signed integer values that are 8, 16, 32, 64, or 128 bits long. Functions in this category compute the output size (in print positions) of an integer, convert an integer to a sequence of characters, and convert a sequence of characters to an integer value.

### 8.4.1 Internal Functions

These functions are for internal use by the standard library. You should not call these functions in your own code. The internal functions in this category are `conv._intToBuf32`, `conv._intToBuf32Size`, `conv._intToBuf64`, `conv._intToBuf64Size`, `conv._intToBuf128`, `conv._intToBuf128Size`

### 8.4.2 Integer Size Calculations

These routines return the size, in screen print positions, it would take to print the signed integer passed in the specified parameter. They return their value in the EAX register (the value always fits in AL and AX, if you'd prefer to use these registers as the return value). The count includes room for a minus sign if the number is negative. Note that these routines include print positions required by underscores if you've enabled underscore output in values (see `conv.setUnderscores` and `conv.getUnderscores` for details).

```
procedure conv.i8Size( b:byte in al ); @returns( "eax" );
```

This function computes the number of print positions required by the 8-bit signed integer passed in the AL register. Because the number of decimal positions is always three or less, the internal underscores flag does not affect the value this function returns. This function will always return a value in the range 1..4 (e.g., four positions for a value like "-128").

HLA high-level calling sequence examples:

```
conv.i8Size( byteVariable );
mov( eax, numSize );

conv.i8Size( <byte register> ); // al, ah, bl, bh, cl, ch, dl, dh
mov( eax, int8Size );

conv.i8Size( <constant> );      // Must fit into eight bits
mov( al, constantsSize );
```

Because `conv.i8Size` passes its input parameter in the AL register, any form of the high-level calling sequence except "`conv.i8Size( al );`" will automatically generate an instruction of the form "`mov(<operand>,al);`". Therefore, if possible, you should try to have the value whose size you wish to compute already sitting in the AL register and pass AL as the parameter to `conv.i8Size`.

HLA low-level calling sequence examples:

```
mov( byteVariable, al );
call conv.i8Size;
mov( eax, numSize );

mov( <byte register>, al ); // ah, bl, bh, cl, ch, dl, dh
call conv.i8Size;
mov( ax, wordVariable );

// Explicit Examples:

mov( bh, al );
call conv.i8Size;
mov( al, bhSize );
```

```

call conv.i8Size; // Assume value is already in AL
mov( al, alSize );

mov( 123, al );    // Example of computing the size of a constant
call conv.i8Size;
mov( eax, constsSize );

```

It might seem silly to compute the size of a constant as this last example is doing, as the constant's print width is known at compile time. Note, however, that this sequence could appear as part of a macro expansion and the literal constant "123" could actually be the result of expanding a macro parameter.

**procedure conv.i16Size( w:word in ax ); @returns( "eax" );**

This function computes the number of print positions required by the 16-bit signed integer passed in the AX register. If the internal underscores flag is set and the integer value is greater than 999 (or less than -999) then this function will account for the underscores injected by the integer to string conversion routines. This function will always return a value in the range 1..6 if the underscores flag is not set (e.g., "-12345") or a value in the range 1...7 if the internal underscores flag is set (e.g., "-12\_345").

HLA high-level calling sequence examples:

```

conv.i16Size( wordVariable );
mov( eax, numSize );

conv.i16Size( <word register> ); // ax, bx, cx, dx, bp, sp, si, di
mov( eax, int16Size );

conv.i16Size( <constant> );      // Must fit into 16 bits
mov( al, constantsSize );

```

Because conv.i16Size passes its input parameter in the AX register, any form of the high-level calling sequence except "conv.i16Size( ax );" will automatically generate an instruction of the form "mov(<operand>,ax);". Therefore, if possible, you should try to have the value whose size you wish to compute already sitting in the AX register and pass AX as the parameter to conv.i16Size.

HLA low-level calling sequence examples:

```

mov( wordVariable, ax );
call conv.i16Size;
mov( eax, numSize );

mov( <word register>, ax ); // bx, cx, dx, bp, sp, si, or di
call conv.i16Size;
mov( ax, wordVariable );

// Explicit Examples:

mov( bx, ax );
call conv.i16Size;
mov( al, bxSize );

call conv.i16Size; // Assume value is already in AX
mov( al, axSize );

mov( 12345, ax );    // Example of computing the size of a constant
call conv.i16Size;
mov( eax, constsSize );

```

See the comment at the end of `conv.i8Size` about passing constants to these functions.

```
procedure conv.i32Size( d:dword in eax ); @returns( "eax" );
```

This function computes the number of print positions required by the 32-bit signed integer passed in the EAX register. If the internal underscores flag is set this function will include print positions for the underscores that the conversion routines will inject into the string. This function will always return a value in the range 1..11 if the underscores flag is not set (e.g., "-1000000000") or a value in the range 1..14 if the internal underscores flag is set (e.g., "-1\_000\_000\_000").

HLA high-level calling sequence examples:

```
conv.i32Size( wordVariable );
mov( eax, numSize );

conv.i32Size( <dword register> ); // eax, ebx, ecx, edx,
mov( eax, int16Size ) // ebp, esp, esi, or edi

conv.i32Size( <constant> );      // Must fit into 32 bits
mov( al, constantsSize );
```

Because `conv.i32Size` passes its input parameter in the EAX register, any form of the high-level calling sequence except "`conv.i32Size( eax );`" will automatically generate an instruction of the form "`mov(<operand>,eax);`". Therefore, if possible, you should try to have the value whose size you wish to compute already sitting in the EAX register and pass EAX as the parameter to `conv.i32Size`.

HLA low-level calling sequence examples:

```
mov( dwordVariable, eax );
call conv.i32Size;
mov( eax, numSize );

mov( <dword register>, eax ); // ebx, ecx, edx,
call conv.i32Size; // ebp, esp, esi, or edi
mov( ax, wordVariable );

// Explicit Examples:

mov( ebx, eax );
call conv.i32Size;
mov( al, bxSize );

call conv.i32Size; // Assume value is already in AX
mov( al, axSize );

mov( 1234567890, eax ); // Example of computing
call conv.i32Size; // the size of a constant.
mov( eax, constsSize );
```

See the comment at the end of `conv.i8Size` about passing constants to these functions.

```
procedure conv.i64Size( q:qword ); @returns( "eax" );
```

This function computes the number of print positions required by the 64-bit signed integer passed on the stack. If the internal underscores flag is set this function will include print positions for the underscores that the conversion routines will inject into the string. This function will always return a value in the range 1..20 if the underscores flag is not set (e.g., "-9223372036854775807") or a value in the range 1..26 if the internal underscores flag is set (e.g., "-9\_223\_372\_036\_854\_775\_808").

HLA high-level calling sequence examples:

```
conv.i64Size( qwordVariable );
mov( eax, numSize );

conv.i64Size( <constant> );      // Must fit into 64 bits
mov( al, constantsSize );
```

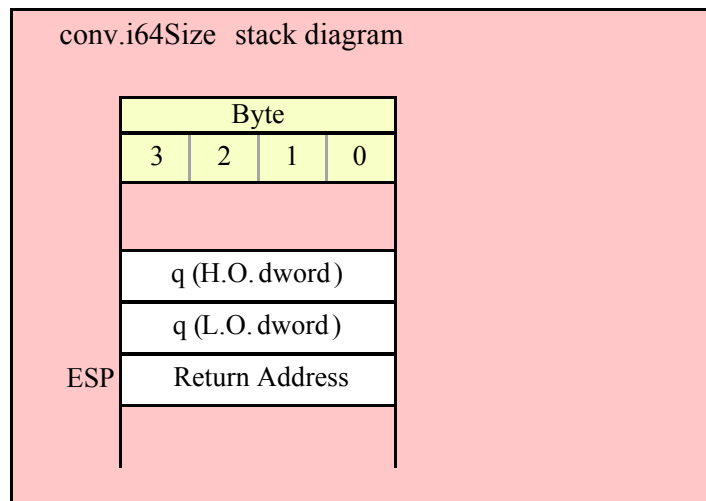
HLA low-level calling sequence examples:

```
push( (type dword qwordVariable[4])); // Push H.O. dword first
push( (type dword qwordVariable[0])); // Push L.O. dword second
call conv.i64Size;
mov( eax, numSize );

// Compute the size of a 64-bit constant:

pushd( 12345 >> 32 ); // Push H.O. dword first
pushd( 12345 & $FFFF_FFFF ); // Push L.O. dword second
call conv.i64Size;
mov( eax, constsSize );
```

See the comment at the end of `conv.i8Size` about passing constants to these functions. If you make a habit of explicitly passing 64-bit constants to this function, you might consider writing a macro to push the 64-bit constant for you (see the chapter on "Passing Parameters to Standard Library Routines" for more details).



```
procedure conv.i128Size( l:1word ); @returns( "eax" );
```

This function computes the number of print positions required by the 128-bit signed integer passed in the EAX register. If the internal underscores flag is set this function will include print positions for the underscores that the conversion routines will inject into the string. This function will always return a value in the range 1..40 if the underscores flag is not set (e.g., "-170141183460469231731687303715884105727") or a value in the range 1...52 if the internal underscores flag is set (e.g., "-170\_141\_183\_460\_469\_231\_731\_687\_303\_715\_884\_105\_728").

HLA high-level calling sequence examples:



```
procedure conv.i8ToBuf( i8 :int8 in al; var buf:var in edi );
```

This function converts the 8-bit signed integer passed in AL to a sequence of 1..4 characters. The string this function produces is always in the range -128..127. Note that because this string always contains three or fewer digits, the internal underscores flag setting does not affect this function's output.

HLA high-level calling sequence examples:

```
// The following will load "byteVariable" into AL and
// the address of "charArrayVariable" into EDI and then
// call conv.i8ToBuf:

conv.i8ToBuf( byteVariable, charArrayVariable );

// The following call will copy BH into AL and
// EDX into EDI prior to calling conv.i8ToBuf:

conv.i8ToBuf( bh, [edx] );

// The following just calls conv.i8ToBuf as AL and EDI
// already hold the parameter values:

conv.i8ToBuf( al, [edi] );

// The following loads the constant in AL and calls
// conv.i8ToBuf:

conv.i8ToBuf( <constant>, [edi] ); // <constant> must fit in 8 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the AL and EDI registers with their respective parameter values. In particular, you should not specify [EAX] as the buffer address because the code that HLA generates can overwrite the L.O. byte of EAX (i.e., AL) before it copies the address to the EDI register. It goes without saying that this function will overwrite the values of EAX and EDI if the original parameters are not AL and [EDI].

HLA low-level calling sequence examples:

```
// Passing a byte variable and a buffer variable:

mov( byteVariable, al );
lea( edi, charArrayVariable );
call conv.i8ToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

mov( byteVariable, al );
mov( &charArrayVariable, edi );
call conv.i8ToBuf;

// Passing a pair of registers (that are not
// AL and EDI):

mov( bh, al );
mov( edx, edi );
call conv.i8ToBuf;

// Passing a constant:

mov( <constant>, al );
```

```
call conv.i18ToBuf; // Assume EDI already contains buffer address.
```

```
procedure conv.i16ToBuf( i16 :int16 in ax; var buf:var in edi )
```

This function converts the 16-bit signed integer passed in AX to a sequence of 1..6 characters if the internal underscores flag is false, 1..7 characters if the underscores flag contains true. The string this function produces is always in the range -32768..32767. If the internal underscores flag contains true and the value is greater than 999 or less than -999, then this function emits an underscore between the third and fourth digits (from the right) in the string.

HLA high-level calling sequence examples:

```
// The following will load "wordVariable" into AX and
// the address of "charArrayVariable" into EDI and then
// call conv.i16ToBuf:

conv.i16ToBuf( wordVariable, charArrayVariable );

// The following call will copy BX into AX and
// EDX into EDI prior to calling conv.i16ToBuf:

conv.i16ToBuf( bx, [edx] );

// The following just calls conv.i16ToBuf as AX and EDI
// already hold the parameter values:

conv.i16ToBuf( ax, [edi] );

// The following loads the constant in AX and calls
// conv.i16ToBuf:

conv.i16ToBuf( <constant>, [edi] ); // <constant> must fit in 16 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the AX and EDI registers with their respective parameter values. In particular, you should not specify [EAX] as the buffer address because the code that HLA generates can overwrite the L.O. word of EAX (i.e., AX) before it copies the address to the EDI register. It goes without saying that this function will overwrite the values of EAX and EDI if the original parameters are not AX and [EDI].

HLA low-level calling sequence examples:

```
// Passing a word variable and a buffer variable:

mov( wordVariable, ax );
lea( edi, charArrayVariable );
call conv.i16ToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

mov( wordVariable, ax );
mov( &charArrayVariable, edi );
call conv.i16ToBuf;

// Passing a pair of registers (that are not
// AX and EDI):

mov( bx, ax );
mov( edx, edi );
```

```

call conv.i16ToBuf;

// Passing a constant:

mov( <constant>, ax );
call conv.i16ToBuf; // Assume EDI already contains buffer address.

```

#### **procedure conv.i32ToBuf( i32 :int32 in eax; var buf:var in edi )**

This function converts the 32-bit signed integer passed in EAX to a sequence of 1..11 characters if the internal underscores flag is false, 1..14 characters if the underscores flag contains true. The string this function produces is always in the range -2147483648..2147483647. If the internal underscores flag contains true and the value is greater than 999 or less than -999, then this function emits an underscore between the third and fourth digits (from the right) in the string.

HLA high-level calling sequence examples:

```

// The following will load "dwordVariable" into EAX and
// the address of "charArrayVariable" into EDI and then
// call conv.i32ToBuf:

conv.i32ToBuf( dwordVariable, charArrayVariable );

// The following call will copy EBX into EAX and
// EDX into EDI prior to calling conv.i32ToBuf:

conv.i32ToBuf( ebx, [edx] );

// The following just calls conv.i32ToBuf as EAX and EDI
// already hold the parameter values:

conv.i32ToBuf( eax, [edi] );

// The following loads the constant in EAX and calls
// conv.i32ToBuf:

conv.i32ToBuf( <constant>, [edi] ); // <constant> must fit in 32 bits

```

When using the HLA high-level calling form, always keep in mind that these statements load the EAX and EDI registers with their respective parameter values. In particular, you should not specify [EAX] as the buffer address because the code that HLA generates can overwrite EAX before it copies the address to the EDI register. It goes without saying that this function will overwrite the values of EAX and EDI if the original parameters are not EAX and [EDI].

HLA low-level calling sequence examples:

```

// Passing a dword variable and a buffer variable:

mov( dwordVariable, eax );
lea( edi, charArrayVariable );
call conv.i32ToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

mov( dwordVariable, eax );
mov( &charArrayVariable, edi );
call conv.i32ToBuf;

```



```
// Passing a pair of registers (that are not
// EAX and EDI):

mov( ebx, eax );
mov( edx, edi );
call conv.i32ToBuf;

// Passing a constant:

mov( <constant>, eax );
call conv.i32ToBuf; // Assume EDI already contains buffer address.
```

#### **procedure conv.i64ToBuf( q :qword; var buf:var in edi )**

This function converts the 64-bit signed integer passed in q to a sequence of 1..20 characters if the internal underscores flag is false, 1..26 characters if the underscores flag contains true. The string this function produces is always in the range -9223372036854775808 .. 9223372036854775807. If the internal underscores flag contains true and the value is greater than 999 or less than -999, then this function emits an underscore between each group of three digits starting with the least significant digit.

HLA high-level calling sequence examples:

```
// The following will push the value of "qwordVariable"
// onto the stack, load the address of "charArrayVariable"
// into EDI and then call conv.i64ToBuf:

conv.i64ToBuf( qwordVariable, charArrayVariable );

// The following pushes the constant onto the stack and calls
// conv.i64ToBuf:

conv.i64ToBuf( <constant>, [edi] ); // <constant> must fit in 64 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the EDI register with the respective parameter value. It goes without saying that this function will overwrite the value of EDI if the original parameter is not [EDI].

HLA low-level calling sequence examples:

```
// Passing a qword variable and a buffer variable:

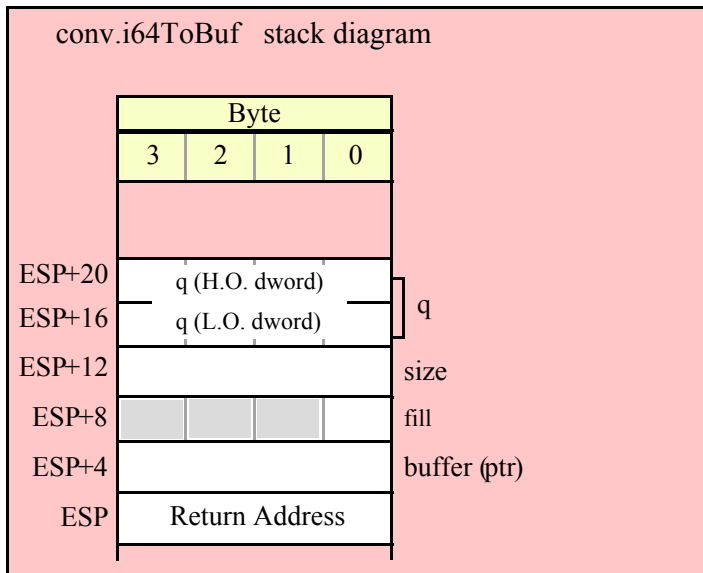
    push( (type dword qwordVariable[4])); // H.O. dword first
    push( (type dword qwordVariable[0])); // L.O. dword last
    lea( edi, charArrayVariable );
    call conv.i64ToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

    push( (type dword qwordVariable[4])); // H.O. dword first
    push( (type dword qwordVariable[0])); // L.O. dword last
    mov( &charArrayVariable, edi );
    call conv.i64ToBuf;

// Passing a constant:
```

```
pushd( <constant> >> 32 );// Push H.O. dword of constant first.
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword second.
call conv.i64ToBuf; // Assume EDI already contains buffer address.
```



```
procedure conv.i128ToBuf( l :lword; var buf:var in edi )
```

This function converts the 128-bit signed integer passed in `l` to a sequence of 1..40 characters if the internal `underscores` flag is false, 1..53 characters if the `underscores` flag contains true. The string this function produces is always in the range -170141183460469231731687303715884105728 .. 170141183460469231731687303715884105727. If the internal `underscores` flag contains true and the value is greater than 999 or less than -999, then this function emits an underscore between each group of three digits starting with the least significant digit.

HLA high-level calling sequence examples:

```
// The following will push the value of "lwordVariable"
// onto the stack, load the address of "charArrayVariable"
// into EDI and then call conv.i128ToBuf:
```

```
conv.i128ToBuf( lwordVariable, charArrayVariable );
```

```
// The following pushes the constant onto the stack and calls
// conv.i128ToBuf:
```

```
conv.i128ToBuf( <constant>, [edi] ); // <constant> must fit in 128 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the EDI register with the respective parameter value. It goes without saying that this function will overwrite the value of EDI if the original parameter is not [EDI].

HLA low-level calling sequence examples:

```
// Passing an lword variable and a buffer variable:
```

```
push( (type dword lwordVariable[12])); // H.O. dword first
```

```

    push( (type dword lwordVariable[8]));
    push( (type dword lwordVariable[4]));
    push( (type dword lwordVariable[0]));// L.O. dword last
    lea( edi, charArrayVariable );
    call conv.i128ToBuf;

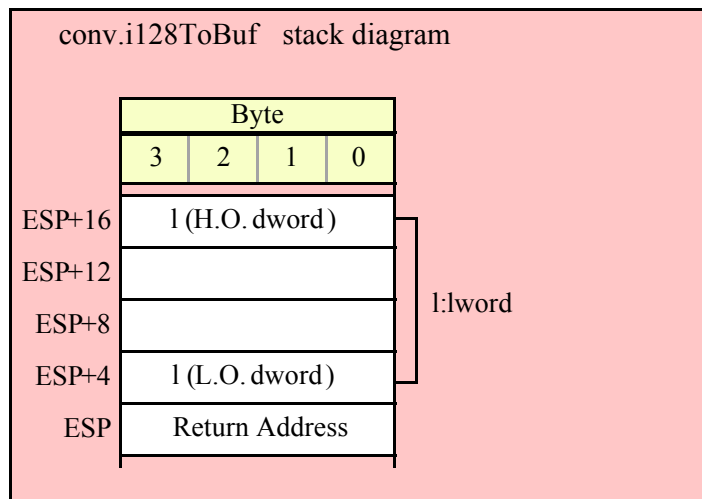
// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

    push( (type dword lwordVariable[12]));// H.O. dword first
    push( (type dword lwordVariable[8]));
    push( (type dword lwordVariable[4]));
    push( (type dword lwordVariable[0]));// L.O. dword last
    mov( &charArrayVariable, edi );
    call conv.i128ToBuf;

// Passing a constant:

    pushd( <constant> >> 96 );// Push H.O. dword of constant first.
    pushd( (<constant> >> 64)& $FFFF_FFFF );
    pushd( (<constant> >> 32)& $FFFF_FFFF );
    pushd( <constant> & $FFFF_FFFF );// Push L.O. dword last.
    call conv.i128ToBuf; // Assume EDI already contains buffer address.

```



### 8.4.4 Integer Numeric to String Conversions

These routines convert a signed integer value ( 8, 16, 32, 64, or 128 bits) to a string. The standard ("unadorned") functions store the string data into a string object that you pass as a parameter to the function. That string object must be preallocated and large enough to receive the string result (else a string overflow occurs). The "adorned" functions, whose names begin with "a\_" automatically allocate storage on the heap, store the converted string into that heap object, and then return a pointer to the newly allocated string in the EAX register (it is the caller's responsibility to free the storage when it is no longer needed).

These functions let you specify a minimum field width and a fill character. If the number would require fewer than width print positions, the routines copy the fill character to the remaining positions in the destination string. If width is positive, the number is right justified in the string. If width is negative, the number is left justified in the string. If the string representation of the value requires more than width print positions, then these functions ignore the width and fill parameters and use however many positions are necessary to properly display the value.

`xxxToStr ( value, width, fill, buffer );`

Assuming “value” requires five print positions, “width” is eight, and fill is “f” then the `xxxToStr` functions produce the string

f	f	f	V	A	L	U	E
---	---	---	---	---	---	---	---

Assuming “value” requires five print positions, “width” is minus eight, and fill is “f” then the `xxxToStr` functions produce the string

V	A	L	U	E	f	f	f
---	---	---	---	---	---	---	---

Here are the maximum number of print positions these routines will produce for each data type before considering the minimum field width:

Underscores flag is false:

```

8 bits:4 (-128..127)
16 bits:6 (-32768..32767)
32 bits:11 (-2147483648..2147483647)
64 bits:20 (-9223372036854775808..9223372036854775807)
128 bits:40 (-170141183460469231731687303715884105728 ..
170141183460469231731687303715884105728)

```

Underscores flag is true:

```

8 bits:4 (-128..127)
16 bits:7 (-32_768..32_767)
32 bits:14 (-2_147_483_648..2_147_483_647)
64 bits:26 (-9_223_372_036_854_775_808..9_223_372_036_854_775_807)
128 bits:52 (-170_141_183_460_469_231_731_687_303_715_884_105_728 ..
170_141_183_460_469_231_731_687_303_715_884_105_728)

```

**`procedure conv.i8ToStr ( b:int8; width:int32; fill:char; dest:string );`**

This function converts an 8-bit signed integer to the decimal string representation of that integer and stores the string in the preallocated string object specified by the `dest` parameter. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). This function will raise a string overflow exception if the destination string is not large enough to hold the conversion. Note that the internal underscores flag will not affect the output because 8-bit integers are always three digits or smaller.

HLA high-level calling sequence examples:

```

// The following will push "byteVariable" and "destStr" (which
// is a pointer to a string object) and then call conv.i8ToStr:

```

```
conv.i8ToStr( byteVariable, destStr );
```

```

// The following call will BH's value onto the stack and then
// push EDX's value (which is the address of a string object)
// before calling conv.i8ToStr:

```

```
conv.i8ToStr( bh, edx );
```

```

// The following pushes the constant and destStr and calls
// conv.i8ToStr:

```

```
conv.bToBuf( <constant>, destStr ); // <constant> must fit in 8 bits
```

When using the HLA high-level calling form, remember that string variables are dword pointers that contain the address of a string object. The destination string parameter is passed by value, not by reference; it just turns out that the value of a string is a pointer to the actual string data (confusing, isn't it?). In any case, this is why you can pass a register value as the destination string location rather than having to pass something like "[edx]". A construct like "[edx]" would imply that EDX contains the address of the string variable, that is, a pointer to the pointer to the string.

HLA low-level calling sequence examples:

```
// Passing a byte variable and a buffer variable, option 1
// (if a 32-bit register is available):

movzx( byteVariable, eax );
push( eax );
push( destStr );
call conv.i8ToStr;

// Passing a byte variable and a buffer variable, option 2
// (if byteVariable isn't the last byte in mapped memory):

push( (type dword byteVariable));
push( destStr );
call conv.i8ToStr;

// Passing a byte variable and a buffer variable, option 3
// No registers are available and we can't guarantee that
// the three bytes following byteVariable are present in
// mapped memory:

sub( 4, esp );
push( eax );
movzx( byteVariable, eax );
mov( eax, [esp+4] );
pop( eax );
push( destStr );
call conv.i8ToStr;

// Passing a pair of registers (hex value in AL, BL, CL, or DL):
// BL = value to print, EDX = pointer to string object.

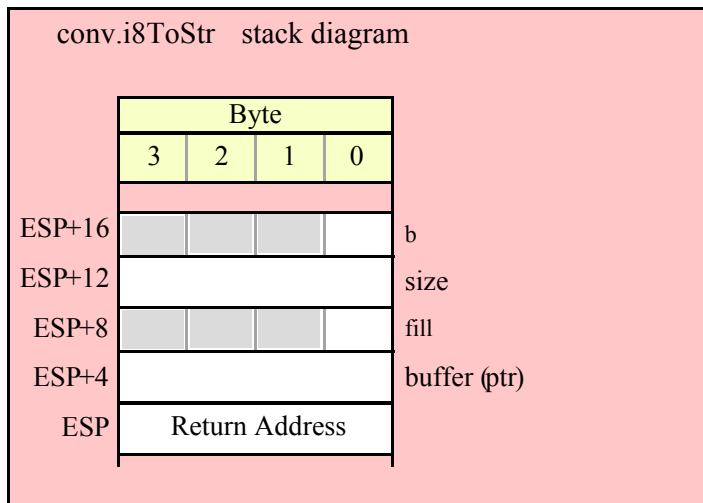
push( ebx );      // Pushes BL
push( edx );
call conv.i8ToStr;

// Passing a pair of registers (hex value in AH, BH, CH, or DH):
// BH = value to print, EDX = pointer to string object.

pushd( 0 );
mov( bh, [esp] );
push( edx );
call conv.i8ToStr;

// Passing a constant:

pushd( <constant> );
push( destStr );
call conv.i8ToStr;
```



```

procedure conv.a_i8ToStr ( b:int8;  width:int32; fill:char );
    @returns( "eax" );

```

This function converts an 8-bit signed integer to the decimal string representation of that integer and stores the string in storage it allocates on the heap. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). The caller is responsible for freeing the storage when it is no longer needed. Note that the internal underscores flag will not affect the output because 8-bit integers are always three digits or smaller.

HLA high-level calling sequence examples:

```
// The following will push "byteVariable" and then call conv.a_i8ToStr:
```

```
conv.a_i8ToStr( byteVariable );
mov( eax, byteStr );
```

```
// The following call will BH's value onto the stack
// before calling conv.a_i8ToStr:
```

```
conv.a_i8ToStr( bh );
mov( eax, byteStr );
```

```
// The following pushes the constant and calls
// conv.a_i8ToStr:
```

```
conv.a_i8ToStr( <constant> ); // <constant> must fit in 8 bits
mov( eax, byteStr );
```

HLA low-level calling sequence examples:

```
// Passing a byte variable and a buffer variable, option 1
// (if a 32-bit register is available):
```

```
movzx( byteVariable, eax );
push( eax );
call conv.a_i8ToStr;
mov( eax, destStr );
```

```

// Passing a byte variable and a buffer variable, option 2
// (if byteVariable isn't the last byte in mapped memory):

push( (type dword byteVariable));
call conv.a_i8ToStr;
mov( eax, destStr );

// Passing a byte variable, option 3
// No registers are available and we can't guarantee that
// the three bytes following byteVariable are present in
// mapped memory:

sub( 4, esp );
push( eax );
movzx( byteVariable, eax );
mov( eax, [esp+4] );
pop( eax );
call conv.a_i8ToStr;
mov( eax, destStr );

// Passing a pair of registers (hex value in AL, BL, CL, or DL):
// BL = value to print, EDX = pointer to string object.

push( ebx );          // Pushes BL
call conv.a_i8ToStr;
mov( eax, byteStr );

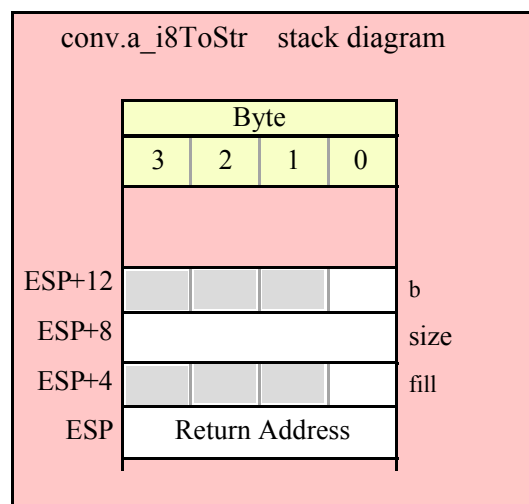
// Passing a pair of registers (hex value in AH, BH, CH, or DH):
// BH = value to print, EDX = pointer to string object.

pushd( 0 );
mov( bh, [esp] );
call conv.a_i8ToStr;
mov( eax, byteStr );

// Passing a constant:

pushd( <constant> );
call conv.a_i8ToStr;
mov( eax, byteStr );

```



```
procedure conv.i16ToStr( w:int16; width:int32; fill:char; dest:string );
```

This function converts a 16-bit signed integer to its decimal string representation and stores the string in the preallocated string object specified by the dest parameter. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). This function will raise a string overflow exception if the destination string is not large enough to hold the conversion. If the conversion requires more than three digits and the internal underscores flag is true, then this function will insert an underscore between each group of three digits, starting with the least significant digit (see conv.getUnderscores and conv.setUnderscores for more details).

HLA high-level calling sequence examples:

```
// The following will push "wordVariable" and "destStr" (which
// is a pointer to a string object) and then call conv.i16ToStr:

conv.i16ToStr( wordVariable, destStr );

// The following call will BX's value onto the stack and then
// push EDX's value (which is the address of a string object)
// before calling conv.i16ToStr:

conv.i16ToStr( bx, edx );

// The following pushes the constant and destStr and calls
// conv.i16ToStr:

conv.i16ToStr( <constant>, destStr ); // <constant> must fit in 16 bits
```

When using the HLA high-level calling form, remember that string variables are dword pointers that contain the address of a string object. The destination string parameter is passed by value, not by reference; it just turns out that the value of a string is a pointer to the actual string data (confusing, isn't it?). In any case, this is why you can pass a register value as the destination string location rather than having to pass something like "[edx]". A construct like "[edx]" would imply that EDX contains the address of the string variable, that is, a pointer to the pointer to the string.

HLA low-level calling sequence examples:

```
// Passing a word variable and a buffer variable, option 1
// (if a 32-bit register is available):

movzx( byteVariable, eax );
push( eax );
push( destStr );
call conv.i16ToStr;

// Passing a word variable and a buffer variable, option 2
// (if wordVariable isn't the last byte in mapped memory):

push( (type dword wordVariable));
push( destStr );
call conv.i16ToStr;

// Passing a word variable and a buffer variable, option 3
// No registers are available and we can't guarantee that
// the two bytes following wordVariable are present in
// mapped memory:
```



```

pushw( 0 );
push( wordVariable );
push( destStr );
call conv.i16ToStr;

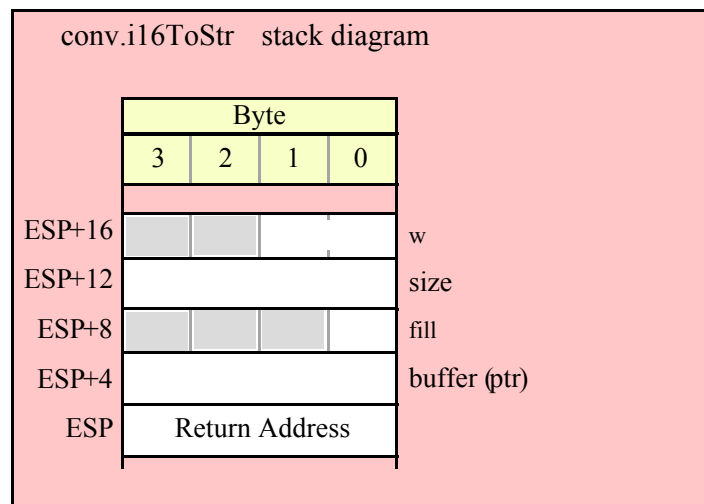
// Passing a pair of registers:
// BX = value to print, EDX = pointer to string object.

push( ebx );      // Pushes BX
push( edx );
call conv.i16ToStr;

// Passing a constant:

pushd( <constant> );
push( destStr );
call conv.i16ToStr;

```



```

procedure conv.a_i16ToStr( w:int16; width:int32; fill:char );
    @returns( "eax" );

```

This function converts a 16-bit signed integer to the decimal string representation of that integer and stores the string in storage it allocates on the heap. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). The caller is responsible for freeing the storage when it is no longer needed. If the conversion produces more than three digits and the internal underscores flag is true, then this function will insert an underscore between each group of three digits, starting with the least significant digit (see conv.getUnderscores and conv.setUnderscores for more details).

HLA high-level calling sequence examples:

```

// The following will push "wordVariable" and call conv.a_i16ToStr:

conv.a_i16ToStr( wordVariable );
mov( eax, destStr );

// The following call will BX's value onto the stack

```

```
// before calling conv.a_i16ToStr:

conv.a_i16ToStr( bx );

// The following pushes the constant and calls
// conv.a_i16ToStr:

conv.a_i16ToStr( <const>, destStr ); // <const> must fit in 16 bits
```

HLA low-level calling sequence examples:

```
// Passing a word variable and a buffer variable, option 1
// (if a 32-bit register is available):
```

```
movzx( wordVariable, eax );
push( eax );
call conv.a_i16ToStr;
mov( eax, destStr );
```

```
// Passing a word variable and a buffer variable, option 2
// (if wordVariable isn't the last byte in mapped memory):
```

```
push( (type dword wordVariable));
call conv.a_i16ToStr;
mov( eax, destStr );
```

```
// Passing a word variable and a buffer variable, option 3
// No registers are available and we can't guarantee that
// the two bytes following wordVariable are present in
// mapped memory:
```

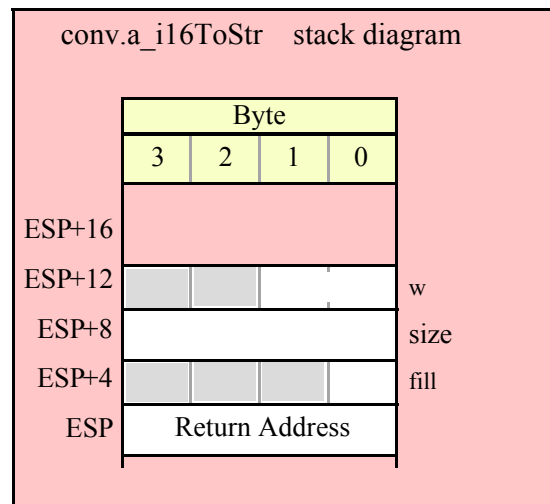
```
pushw( 0 );
push( wordVariable );
call conv.a_i16ToStr;
mov( eax, destStr );
```

```
// Passing a pair of registers:
// BX = value to print.
```

```
push( ebx );          // Pushes BX
call conv.a_i16ToStr;
mov( eax, wordStr );
```

```
// Passing a constant:
```

```
pushd( <constant> );
call conv.a_i16ToStr;
mov( eax, destStr );
```



```
procedure conv.i32ToStr( d:int32; width:int32; fill:char; buffer:string );
```

This function converts a 32-bit signed integer to its decimal string representation and stores the string in the preallocated string object specified by the buffer parameter. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). This function will raise a string overflow exception if the destination string is not large enough to hold the conversion. If the conversion requires more than three digits and the internal underscores flag is true, then this function will insert an underscore between each group of three digits, starting with the least significant digit (see conv.getUnderscores and conv.setUnderscores for more details).

HLA high-level calling sequence examples:

```
// The following will push "dwordVariable" and "destStr" (which
// is a pointer to a string object) and then call conv.i32ToStr:

conv.i32ToStr( dwordVariable, destStr );

// The following call will push EBX's value onto the stack and then
// push EDX's value (which is the address of a string object)
// before calling conv.i32ToStr:

conv.i32ToStr( ebx, edx );

// The following pushes the constant and destStr and calls
// conv.i32ToStr:

conv.i32ToStr( <constant>, destStr ); // <constant> must fit in 32 bits
```

When using the HLA high-level calling form, remember that string variables are dword pointers that contain the address of a string object. The destination string parameter is passed by value, not by reference; it just turns out that the value of a string is a pointer to the actual string data. In any case, this is why you can pass a register value as the destination string location rather than having to pass something like "[edx]". A construct like "[edx]" would imply that EDX contains the address of the string variable, that is, a pointer to the pointer to the string data.

HLA low-level calling sequence examples:

```
// Passing a dword variable and a buffer variable:

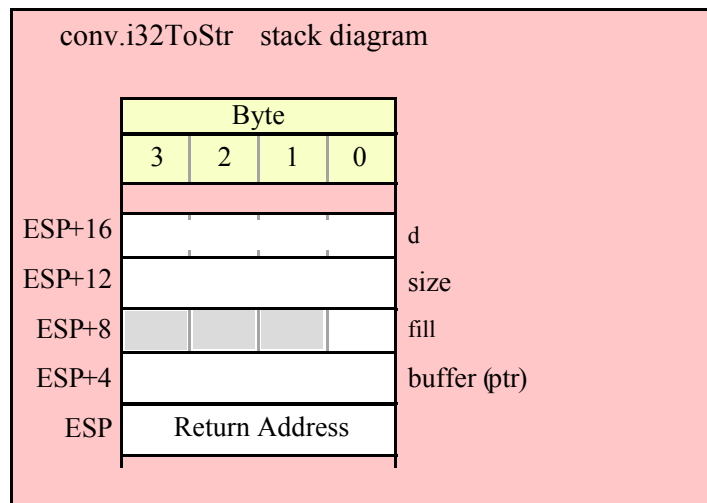
push( dwordVariable );
push( destStr );
call conv.i32ToStr;

// Passing a pair of registers:
// EBX = value to print, EDX = pointer to string object.

push( ebx );
push( edx );
call conv.i32ToStr;

// Passing a constant:

pushd( <constant> );
push( destStr );
call conv.i32ToStr;
```



```
procedure conv.a_i32ToStr( d:int32; width:int32; fill:char );
    @returns( "eax" );
```

This function converts a 32-bit signed integer to the decimal string representation of that integer and stores the string in storage it allocates on the heap. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). The caller is responsible for freeing the storage when it is no longer needed. If the conversion produces more than three digits and the internal underscores flag is true, then this function will insert an underscore between each group of three digits, starting with the least significant digit (see `conv.getUnderscores` and `conv.setUnderscores` for more details).

HLA high-level calling sequence examples:

```
// The following will push "dwordVariable" and then call conv.a_i32ToStr:

conv.a_i32ToStr( dwordVariable );
mov( eax, destStr );
```

```
// The following call will push EBX's value onto the stack
// before calling conv.a_i32ToStr:

conv.a_i32ToStr( ebx );

// The following pushes the constant and calls
// conv.a_i32ToStr:

conv.a_i32ToStr( <constant>, destStr ); // <constant> must fit in 32 bits
```

HLA low-level calling sequence examples:

```
// Passing a dword variable:
```

```
push( dwordVariable );
call conv.a_i32ToStr;
mov( eax, destStr );
```

```
// Passing a register:
// EBX = value to print.
```

```
push( ebx );
call conv.a_i32ToStr;
mov( eax, dwordStr );
```

```
// Passing a constant:
```

```
pushd( <constant> );
call conv.a_i32ToStr;
mov( eax, destStr );
```

**procedure conv.i64ToStr( q:qword; width:int32; fill:char; buffer:string );**

This function converts a 64-bit signed integer to its decimal string representation and stores the string in the preallocated string object specified by the buffer parameter. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). This function will raise a string overflow exception if the destination string is not large enough to hold the conversion. If the conversion requires more than three digits and the internal underscores flag is true, then this function will insert an underscore between each group of three digits, starting with the least significant digit (see conv.getUnderscores and conv.setUnderscores for more details).

HLA high-level calling sequence examples:

```
// The following will push the value of "qwordVariable"
// and the value of the destStr string variable
// onto the stack and then call conv.i64ToStr:
```

```
conv.i64ToStr( qwordVariable, destStr );
```

```
// The following pushes the constant onto the stack along with
// the value held in the destStr variable and calls
// conv.i64ToStr:
```



```

mov( eax, destStr );

// The following pushes the constant onto the stack and calls
// conv.a_i64ToStr:

conv.a_i64ToStr( <constant> ); // <constant> must fit in 64 bits
mov( eax, destStr );

```

HLA low-level calling sequence examples:

```

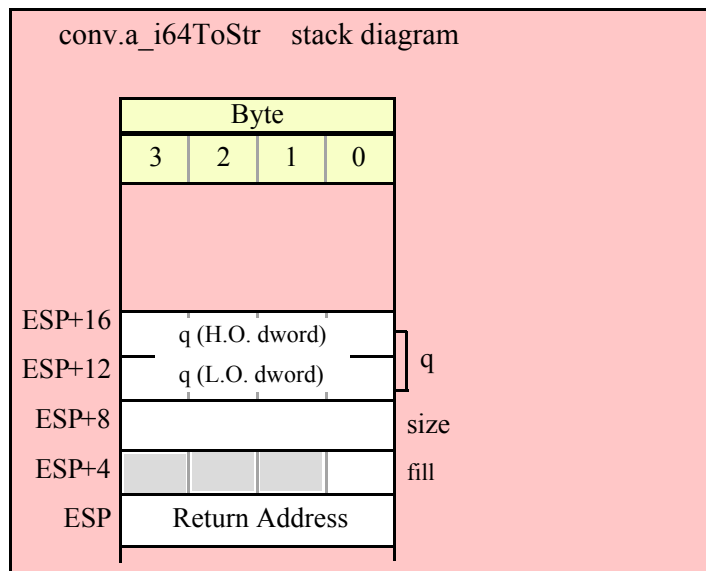
// Passing a qword variable:

    push( (type dword qwordVariable[4])); // H.O. dword first
    push( (type dword qwordVariable[0])); // L.O. dword last
call conv.a_i64ToStr;
mov( eax, destStr );

// Passing a constant:

pushd( <constant> >> 32 ); // Push H.O. dword of constant first.
pushd( <constant> & $FFFF_FFFF ); // Push L.O. dword second.
call conv.a_i64ToStr;
mov( eax, destStr );

```



```
procedure conv.i128ToStr( l:1word; width:int32; fill:char; buffer:string );
```

This function converts a 128-bit signed integer to its decimal string representation and stores the string in the preallocated string object specified by the buffer parameter. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). This function will raise a string overflow exception if the destination string is not large enough to hold the conversion. If the conversion requires more than three digits and the internal underscores flag

is true, then this function will insert an underscore between each group of three digits, starting with the least significant digit (see `conv.getUnderscores` and `conv.setUnderscores` for more details).

HLA high-level calling sequence examples:

```
// The following will push the value of "lwordVariable"
// and destStr onto the stack, and then call conv.i128ToStr:
```

```
conv.i128ToStr( lwordVariable, destStr );
```

```
// The following pushes the constant onto the stack and calls
// conv.i128ToStr:
```

```
conv.i128ToStr( <constant>, edx ); // EDX contains string pointer value.
```

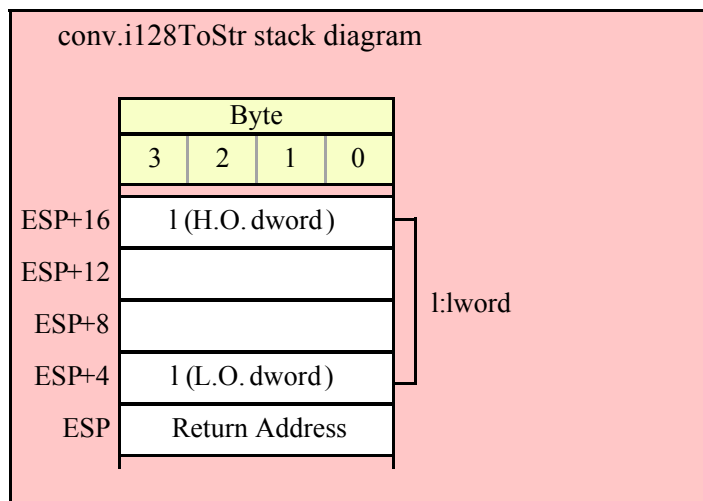
HLA low-level calling sequence examples:

```
// Passing an lword variable:
```

```
    push( (type dword lwordVariable[12])); // H.O. dword first
    push( (type dword lwordVariable[8]));
    push( (type dword lwordVariable[4]));
    push( (type dword lwordVariable[0])); // L.O. dword last
push( destStr );
call conv.i128ToStr;
```

```
// Passing a constant:
```

```
pushd( <constant> >> 96 ); // Push H.O. dword of constant first.
pushd( (<constant> >> 64) & $FFFF_FFFF );
pushd( (<constant> >> 32) & $FFFF_FFFF );
pushd( <constant> & $FFFF_FFFF ); // Push L.O. dword last.
push( edx ); // EDX contains string pointer value.
call conv.i128ToStr;
```





```

procedure conv.a_i128ToStr( l:lword; width:int32; fill:char );
    @returns( "eax" );

```

This function converts a 128-bit signed integer to the decimal string representation of that integer and stores the string in storage it allocates on the heap. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). The caller is responsible for freeing the storage when it is no longer needed. If the conversion produces more than three digits and the internal underscores flag is true, then this function will insert an underscore between each group of three digits, starting with the least significant digit (see conv.getUnderscores and conv.setUnderscores for more details).

HLA high-level calling sequence examples:

```

// The following will push the value of "lwordVariable"
// onto the stack, and then call conv.a_i128ToStr:

conv.a_i128ToStr( lwordVariable );
mov( eax, destStr );

// The following pushes the constant onto the stack and calls
// conv.a_i128ToStr:

conv.a_i128ToStr( <constant> );
mov( eax, destStr );

```

HLA low-level calling sequence examples:

```

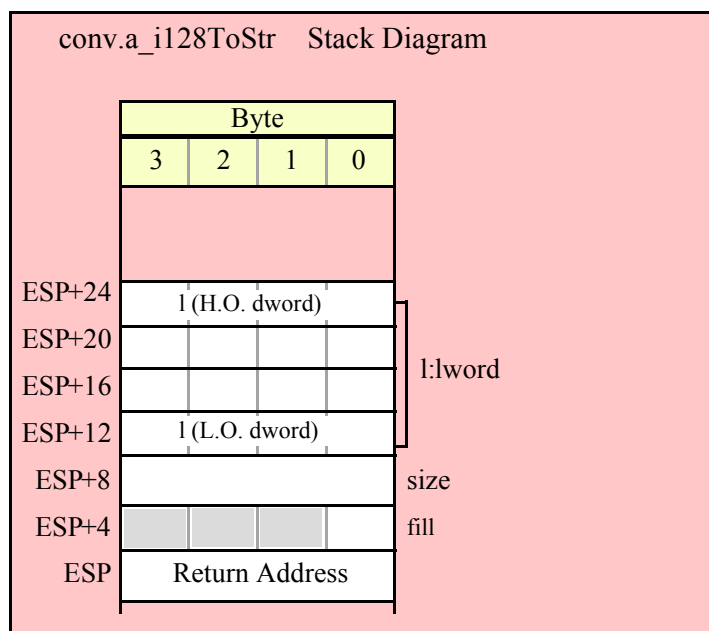
// Passing an lword variable:

    push( (type dword lwordVariable[12])); // H.O. dword first
    push( (type dword lwordVariable[8]));
    push( (type dword lwordVariable[4]));
    push( (type dword lwordVariable[0])); // L.O. dword last
    call conv.a_i128ToStr;
    mov( eax, destStr );

// Passing a constant:

    pushd( <constant> >> 96 ); // Push H.O. dword of constant first.
    pushd( (<constant> >> 64) & $FFFF_FFFF );
    pushd( (<constant> >> 32) & $FFFF_FFFF );
    pushd( <constant> & $FFFF_FFFF ); // Push L.O. dword last.
    call conv.a_i128ToStr;
    mov( eax, destStr );

```



### 8.4.5 Signed Integer String to Numeric Conversions

The standard library string to integer conversion routines convert a sequence of digits, possibly prefaced by a minus sign, into the corresponding signed integer value. These routines begin by skipping over any leading delimiter characters (see the `conv.getDelimiters` and `conv.setDelimiters` functions for details), handling an optional minus sign, followed by any number of decimal digits and underscores (these routines ignore the underscores). Conversion stops at the end of the string or upon encountering a delimiter character.

These routines will raise a conversion error exception if they encounter a 7-bit ASCII character that is not a decimal digit, an underscore, or a delimiter character during the translation. These routines will raise an illegal character exception if they encounter a non-ASCII character (one with its H.O. bit set). These routines will raise a value out of range exception if the converted value will not fit in the destination data object.

There are two basic sets of string to numeric conversion routines: the `conv.atoi*` routines and the `conv.strToi*` routines. The `atoi*` routines process the characters pointed at by the ESI register. The `strToi*` routines process data in a string object, starting at an offset specified by a second parameter. For example, "`conv.strToi8( "12345", 3);`" returns the value 45 because it begins processing the string at (zero-based) offset 3 in the string.

```
procedure conv.atoi8 ( buffer:var in esi ); @returns( "al" );
```

This function converts the sequence of characters starting at the memory address held in ESI to an 8-bit signed integer. It returns the value (in the range -128..+127) in AL. Note that this function actually returns the sign-extended value in EAX, so you may use EAX if it is more convenient to do so. Upon successful return, ESI is pointing at the delimiter character at the end of the sequence of digits.

HLA high-level calling sequence examples:

```
// The following will convert the decimal characters in memory
// at the address specified by [esi] into a numeric value
// and return that value in AL:
```

```
conv.atoi8( [esi] );
mov( al, numericResult );
```

```
// The following loads ESI with the address of
// a sequence of decimal characters (held in an HLA
// string) and converts them to an 8-bit number:

conv.atoi8( sourceStr ); // Loads "sourceStr" into ESI
mov( al, byteVariable );
```

HLA low-level calling sequence examples:

```
// Same as first example above - ESI already contains
// the address of the first character to convert:
```

```
call conv.atoi8;
mov( al, numericResult );
```

```
// Same as second example above
```

```
static
  sourceStr :string := "12";
  .
  .
  .
mov( sourceStr, esi );
call conv.atoi8;
mov( al, num12 );
```

**procedure conv.atoi16 ( buffer:var in esi ); @returns( "ax" );**

This function converts the sequence of characters starting at the memory address held in ESI to a 16-bit signed integer. It returns the value (in the range -32768..+32767) in AX. Note that this function actually returns the sign-extended value in EAX, so you may use EAX if it is more convenient to do so. Upon successful return, ESI is pointing at the delimiter character at the end of the sequence of digits.

HLA high-level calling sequence examples:

```
// The following will convert the decimal characters in memory
// at the address specified by [esi] into a numeric value
// and return that value in AX:
```

```
conv.atoi16( [esi] );
mov( ax, numericResult );
```

```
// The following loads ESI with the address of
// a sequence of decimal characters (held in an HLA
// string) and converts them to a 16-bit number:
```

```
conv.atoi16( sourceStr ); // Loads "sourceStr" into ESI
mov( ax, wordVariable );
```

HLA low-level calling sequence examples:

```
// Same as first example above - ESI already contains
// the address of the first character to convert:
```

```

    call conv.atoi16;
    mov( ax, numericResult );

```

```

// Same as second example above

```

```

static
    sourceStr :string := "12";
    .
    .
    .
mov( sourceStr, esi );
    call conv.atoi16;
    mov( ax, num12 );

```

**procedure conv.atoi32 ( buffer:var in esi ); @returns( "eax" );**

This function converts the sequence of characters starting at the memory address held in ESI to a 32-bit signed integer. It returns the value (in the range -2147483648..+2147483647) in EAX. ESI is pointing at the delimiter character at the end of the sequence of digits.

HLA high-level calling sequence examples:

```

// The following will convert the hexadecimal characters in memory
// at the address specified by [esi] into a numeric value
// and return that value in EAX:

```

```

conv.atoi32( [esi] );
mov( eax, numericResult );

```

```

// The following loads ESI with the address of
// a sequence of decimal characters (held in an HLA
// string) and converts them to a 32-bit number:

```

```

conv.atoi32( sourceStr ); // Loads "sourceStr" into ESI
mov( eax, dwordVariable );

```

HLA low-level calling sequence examples:

```

// Same as first example above - ESI already contains
// the address of the first character to convert:

```

```

    call conv.atoi32;
    mov( eax, numericResult );

```

```

// Same as second example above

```

```

static
    sourceStr :string := "12";
    .
    .
    .
mov( sourceStr, esi );
    call conv.atoi32;
    mov( eax, num12 );

```

```
procedure conv.atoi64 ( buffer:var in esi ); @returns( "edx:eax" );
```

This function converts the sequence of characters starting at the memory address held in ESI to a 64-bit signed integer. It returns the value (in the range -9223372036854775808..+9223372036854775807) in EDX:EAX (EDX contains the H.O. dword). ESI is pointing at the delimiter character at the end of the sequence of digits.

HLA high-level calling sequence examples:

```
// The following will convert the decimal characters in memory
// at the address specified by [esi] into a numeric value
// and return that value in EDX:EAX:
```

```
conv.atoi64( [esi] );
mov( eax, (type dword hex64NumericResult[0]) );
mov( edx, (type dword hex64NumericResult[4]) );
```

```
// The following loads ESI with the address of
// a sequence of decimal characters (held in an HLA
// string) and converts them to a 64-bit number:
```

```
conv.atoi64( sourceStr ); // Loads "sourceStr" into ESI
mov( eax, (type dword qwordVariable[0]) );
mov( edx, (type dword qwordVariable[4]) );
```

HLA low-level calling sequence examples:

```
// Same as first example above - ESI already contains
// the address of the first character to convert:
```

```
call conv.atoi64;
mov( eax, (type dword numericResult[0]) );
mov( edx, (type dword numericResult[4]) );
```

```
// Same as second example above
```

```
static
  sourceStr:string := "12";
.
.
.
mov( sourceStr, esi );
call conv.atoi64;
mov( eax, (type dword qwordVariable[0]) );
mov( edx, (type dword qwordVariable[4]) );
```

```
procedure conv.atoi128( buffer:var in esi; var l:ldword );
```

This function converts the sequence of characters starting at the memory address held in ESI to a 128-bit signed integer. It returns the value (in the range -170141183460469231731687303715884105728..+170141183460469231731687303715884105727) in the l

parameter that is passed by reference to this function. ESI is pointing at the delimiter character at the end of the sequence of digits.

HLA high-level calling sequence examples:

```
// The following will convert the decimal characters in memory
// at the address specified by [esi] into a numeric value
// and stores that value in lwordDest (passed by reference):

conv.atoi128( [esi], lwordDest );

// The following loads ESI with the address of
// a sequence of decimal characters (held in an HLA
// string) and converts them to a 128-bit number that it
// stores in lwordDest:

conv.atoi128( sourceStr, lwordDest ); // Loads "sourceStr" into ESI
```

HLA low-level calling sequence examples:

```
// Option 1: lwordDest is a static object declared in a
// HLA STATIC, READONLY, or STORAGE section:
// As with the first example above, assume ESI already
// contains the address of the string to convert:

pushd( &lwordDest ); // Pass address of lwordDest as reference parm.
call conv.atoi128;

// Option 2: lwordDest is a simple automatic variable (no indexing)
// declared in a VAR section (or as a parameter). Assume that
// no 32-bit registers can be disturbed by this code.
// As with the first example above, assume ESI already
// contains the address of the string to convert:

push( ebp );
add( @offset( lwordDest ), (type dword [esp]));
call conv.atoi128;

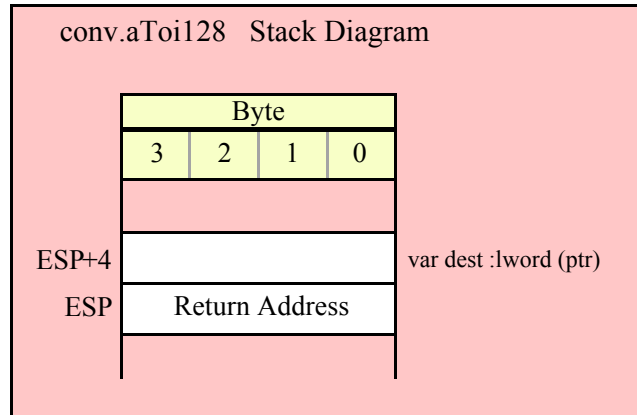
// Option 3: lwordDest is a complex addressing mode and at least
// one 32-bit register is available for use by this code.
// As with the first example above, assume ESI already
// contains the address of the string to convert:

lea( eax, lwordDest ); // Assume EAX is the available register
push( eax );
call conv.atoi128;

// Same as second high-level example above. Assumes that
// lwordDest is a static object.

static
  sourceStr :string := "12";
  .
  .
  .
mov( sourceStr, esi );
pushd( &lwordDest );
```

```
call conv.atoi128;
```



```
procedure conv.strToi8( s:string; index:dword )
```

This function converts the sequence of characters starting at zero-based offset index within the string parameter *s* to an 8-bit signed integer. It returns the value (in the range -128..+127) in AL. Note that this function actually returns the sign-extended value in EAX, so you may use EAX if it is more convenient to do so.

HLA high-level calling sequence examples:

```
// The following will convert the characters at the beginning
// of "decValueStr" to numeric form:
```

```
conv.strToi8( decValueStr, 0 );// Index=0 starts at beginning
mov( al, numericResult );
```

```
// The following demonstrates using a non-zero index:
```

```
conv.strToi8( "abc12", 3 ); // "12" begins at offset 3
mov( al, hex12 );
```

HLA low-level calling sequence examples:

```
push( decValueStr );// Same as first example above
pushd( 0 );
call conv.strToi8;
mov( al, decNumericResult );
```

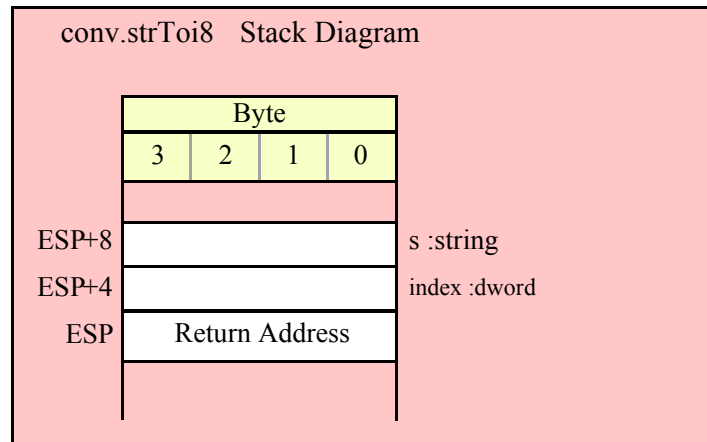
```
// Same as second example above
```

```
static
  str12 :string := "abc12";
  .
  .
  .
push( str12 );// Note that str12 points at "abc12".
```

```

pushd( 3 );// Index to "12" in "abc12".
call conv.strToi8;
mov( al, dec12 );

```



```
procedure conv.strToi16( s:string; index:dword )
```

This function converts the sequence of characters starting at zero-based offset `index` within the string parameter `s` to a 16-bit signed integer. It returns the value (in the range -32768..+32767) in `AX`. Note that this function actually returns the sign-extended value in `EAX`, so you may use `EAX` if it is more convenient to do so.

HLA high-level calling sequence examples:

```
// The following will convert the characters at the beginning
// of "decValueStr" to numeric form:
```

```
conv.strToi16( hexValueStr, 0 );// Index=0 starts at beginning
mov( ax, wordVar );
```

```
// The following demonstrates using a non-zero index:
```

```
conv.strToi16( "abc1234", 3 ); // "1234" begins at offset 3
mov( ax, wordVar );
```

HLA low-level calling sequence examples:

```
push( decValueStr );// Same as first example above
pushd( 0 );
call conv.strToi16;
mov( ax, wordVar );
```

```
// Same as second example above
```

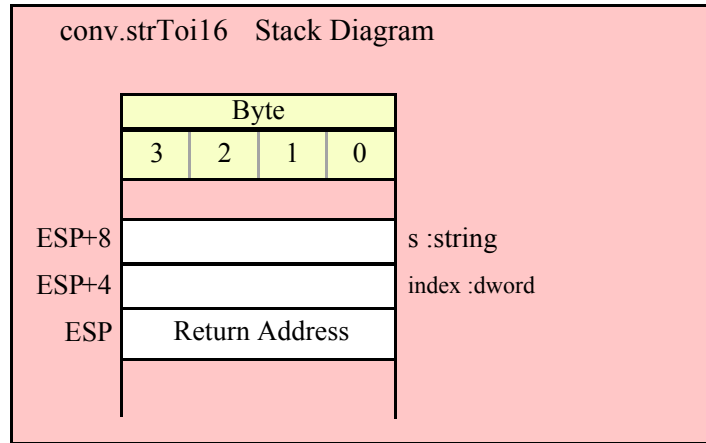
```
static
  str1200 :string := "abc1200";
  .
  .
```



```

push( str1200 );// Note that str1200 points at "abc1200".
pushd( 3 );// Index to "1200" in "abc1200".
call conv.strToi16;
mov( ax, wordVar );

```



#### **procedure conv.strToi32( s:string; index:dword )**

This function converts the sequence of characters starting at zero-based offset index within the string parameter s to a 32-bit signed integer. It returns the value (in the range -2147483648..+2147483647) in EAX.

HLA high-level calling sequence examples:

```

// The following will convert the characters at the beginning
// of "decValueStr" to numeric form:

```

```

conv.strToi32( decValueStr, 0 );// Index=0 starts at beginning
mov( eax, dwordVar );

```

```

// The following demonstrates using a non-zero index:

```

```

conv.strToi32( "abc12_345", 3 ); // "12_345" begins at offset 3
mov( eax, dwordVar );

```

HLA low-level calling sequence examples:

```

push( decValueStr );// Same as first example above
pushd( 0 );
call conv.strToi32;
mov( eax, dwordVar );

```

```

// Same as second example above

```

```

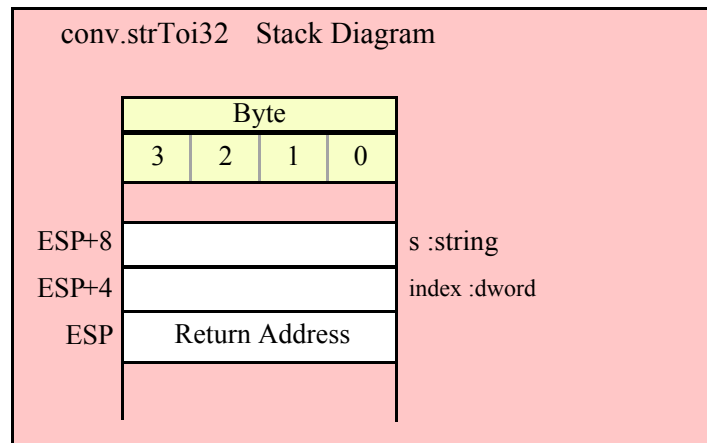
static
  str12345 :string := "abc-12_345";

```

```

push( str12345 );// Note that str12345 points at "abc-12_345".
pushd( 3 ); // Index to "-12_345" in "abc-12_345".
call conv.strToi32;
mov( eax, dwordVar );// dwordVar now contains -12,345.

```



#### **procedure conv.strToi64( s:string; index:dword )**

This function converts the sequence of characters starting at zero-based offset index within the string parameter `s` to a 64-bit signed integer. It returns the value (in the range -9223372036854775808 .. +9223372036854775807) in `EDX:EAX` (`EDX` contains the H.O. dword).

HLA high-level calling sequence examples:

```

// The following will convert the characters at the beginning
// of "decValueStr" to numeric form:

```

```

conv.strToi64( decValueStr, 0 );// Index=0 starts at beginning
mov( eax, (type dword qwordVar[0]) );
mov( edx, (type dword qwordVar[4]) );

```

// The following demonstrates using a non-zero index:

```

conv.strToi64( "a-123", 1 ); // "-123" begins at offset 1
mov( eax, (type dword qwordVar[0]) );
mov( edx, (type dword qwordVar[4]) );

```

HLA low-level calling sequence examples:

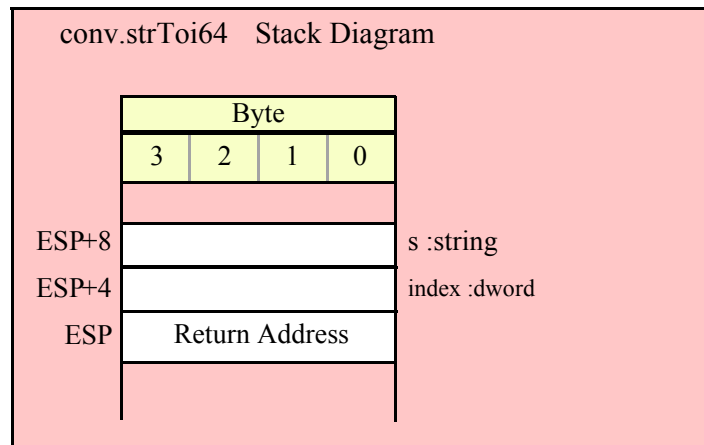
```

push( decValueStr );// Same as first example above
pushd( 0 );
call conv.strToi64;
mov( eax, (type dword qwordVar[0]) );
mov( edx, (type dword qwordVar[4]) );

```

```
// Same as second example above

static
  strabc12 :string := "a-123";
  .
  .
  .
push( strabc12 );// Note that strabc12 points at "a-123".
pushd( 1 ); // Index to "-123" in "a-123".
call conv.strToi64;
mov( eax, (type dword qwordVar[0]) );
mov( edx, (type dword qwordVar[4]) );
```



```
procedure conv.strToi128( s:string; index:dword; var dest:lword )
```

This function converts the sequence of characters starting at zero-based offset index within the string parameter *s* to a 128-bit signed integer. It returns the value (in the range -170141183460469231731687303715884105728..+170141183460469231731687303715884105727) in the parameter *l* that you pass by reference to this function.

HLA high-level calling sequence examples:

```
// The following will convert the characters at the beginning
// of "decValueStr" (index=0) to numeric form and store the
// 128-bit result into the lwordDest variable:
```

```
conv.strToi128( decValueStr, 0, lwordDest );
```

```
// The following demonstrates using a non-zero index:
```

```
conv.strToi128( "abc1234567890123456789", 3, lwordDest );
```

HLA low-level calling sequence examples:

```
// Option #1: lwordDest is a STATIC/READONLY/STORAGE
// variable:
```

```
push( decValueStr );// Same as first example above
```

```

pushd( 0 );
pushd( &lwordDest );
call conv.strToi128;

// Option #2: lwordDest is not a static object and
// a 32-bit register is available for use:

push( decValueStr );// Same as first example above
pushd( 0 );
lea( eax, lwordDest ); // Assuming EAX is available
push( eax );
call conv.strToi128;

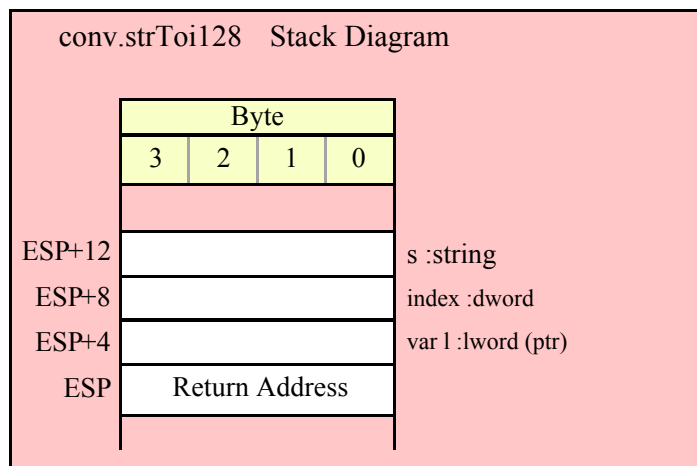
// Option #3: lwordDest is an automatic (var) object and
// no 32-bit registers are available for use:

push( decValueStr );// Same as first example above
pushd( 0 );
push( ebp );
add( @offset( lwordDest ), (type dword [esp]) );
call conv.strToi128;

// Option #4: lwordDest is a complex addressing mode object and
// no 32-bit registers are available for use:

push( decValueStr );// Same as first example above
pushd( 0 );
sub( 4, esp );
push( eax );
lea( eax, lwordDest );
mov( eax, [esp+4] );
pop( eax );
call conv.strToi128;

```



## 8.5 Unsigned Integer Conversions

The integer conversion function process signed integer values that are 8, 16, 32, 64, or 128 bits long. Functions in this category compute the output size (in print positions) of an integer, convert an integer to a sequence of characters, and convert a sequence of characters to an integer value.

## 8.5.1 Internal Routines

The following routines are used internally by the standard library unsigned integer code and you should not directly call them: `conv._u8Size`, `conv._u16Size`, and `conv._u32Size`.

## 8.5.2 Unsigned Integer Size Calculations

These routines return the size, in screen print positions, it would take to print the unsigned integer passed in the specified parameter. They return their value in the EAX register. Note that these routines include print positions required by underscores if you've enabled underscore output in values (see `conv.setUnderscores` and `conv.getUnderscores` for details).

It should go without saying that if you compute the size of an unsigned integer and then change the value of the internal underscores flag, the size you've computed may be invalid.

**procedure `conv.u8Size( b:byte in al ); @returns( "eax" );`**

Computes the output size of an 8-bit unsigned integer (passed in AL) and returns this value in EAX. The return result will always be a value in the range 1..3. The internal underscores flag does not affect the result this function returns.

HLA high-level calling sequence examples:

```
conv.u8Size( byteVariable );
mov( eax, numSize );

conv.u8Size( <byte register> ); // al, ah, bl, bh, cl, ch, dl, dh
mov( eax, int8Size );

conv.u8Size( <constant> );      // Must fit into eight bits
mov( al, constantsSize );
```

Because `conv.u8Size` passes its input parameter in the AL register, any form of the high-level calling sequence except "`conv.u8Size( al );`" will automatically generate an instruction of the form "`mov(<operand>,al);`". Therefore, if possible, you should try to have the value whose size you wish to compute already sitting in the AL register and pass AL as the parameter to `conv.u8Size`.

HLA low-level calling sequence examples:

```
mov( byteVariable, al );
call conv.u8Size;
mov( eax, numSize );

mov( <byte register>, al ); // ah, bl, bh, cl, ch, dl, dh
call conv.u8Size;
mov( ax, wordVariable );

// Explicit Examples:

mov( bh, al );
call conv.u8Size;
mov( al, bhSize );

call conv.u8Size; // Assume value is already in AL
mov( al, alSize );

mov( 123, al );    // Example of computing the size of a constant
```

```
call conv.u8Size;
mov( eax, constsSize );
```

It might seem silly to compute the size of a constant as this last example is doing, as the constant's print width is known at compile time. Note, however, that this sequence could appear as part of a macro expansion and the literal constant "123" could actually be the result of expanding a macro parameter.

#### **procedure conv.ul6Size( w:word in ax )**

Computes the output size of a 16-bit unsigned integer (passed in AX) and returns this value in EAX. The return result will always be a value in the range 1..5 if the internal underscores flag contains false, 1..6 if the underscores flag contains true.

HLA high-level calling sequence examples:

```
conv.ul6Size( wordVariable );
mov( eax, numSize );

conv.ul6Size( <word register> ); // ax, bx, cx, dx, bp, sp, si, di
mov( eax, int16Size );

conv.ul6Size( <constant> );      // Must fit into 16 bits
mov( al, constantsSize );
```

Because conv.ul6Size passes its input parameter in the AX register, any form of the high-level calling sequence except "conv.ul6Size( ax );" will automatically generate an instruction of the form "mov(<operand>,ax);". Therefore, if possible, you should try to have the value whose size you wish to compute already sitting in the AX register and pass AX as the parameter to conv.ul6Size.

HLA low-level calling sequence examples:

```
mov( wordVariable, ax );
call conv.ul6Size;
mov( eax, numSize );

mov( <word register>, ax ); // bx, cx, dx, bp, sp, si, or di
call conv.ul6Size;
mov( ax, wordVariable );

// Explicit Examples:

mov( bx, ax );
call conv.ul6Size;
mov( al, bxSize );

call conv.ul6Size; // Assume value is already in AX
mov( al, axSize );

mov( 12345, ax );    // Example of computing the size of a constant
call conv.ul6Size;
mov( eax, constsSize );
```

See the comment at the end of conv.i8Size about passing constants to these functions.

**procedure conv.u32Size( d:dword in eax )**

Computes the output size of a 32-bit unsigned integer (passed in EAX) and returns this value in EAX. The return result will always be a value in the range 1..10 if the internal underscores flag contains false, 1..11 if the underscores flag contains true.

HLA high-level calling sequence examples:

```
conv.u32Size( wordVariable );
mov( eax, numSize );

conv.u32Size( <dword register> ); // eax, ebx, ecx, edx,
mov( eax, int16Size ) // ebp, esp, esi, or edi

conv.u32Size( <constant> );      // Must fit into 32 bits
mov( al, constantsSize );
```

Because conv.u32Size passes its input parameter in the EAX register, any form of the high-level calling sequence except "conv.u32Size( eax );" will automatically generate an instruction of the form "mov(<operand>,eax);". Therefore, if possible, you should try to have the value whose size you wish to compute already sitting in the EAX register and pass EAX as the parameter to conv.u32Size.

HLA low-level calling sequence examples:

```
mov( dwordVariable, eax );
call conv.u32Size;
mov( eax, numSize );

mov( <dword register>, eax ); // ebx, ecx, edx,
call conv.u32Size; // ebp, esp, esi, or edi
mov( ax, wordVariable );

// Explicit Examples:

mov( ebx, eax );
call conv.u32Size;
mov( al, bxSize );

call conv.u32Size; // Assume value is already in AX
mov( al, axSize );

mov( 1234567890, eax ); // Example of computing
call conv.u32Size; // the size of a constant.
mov( eax, constsSize );
```

See the comment at the end of conv.u8Size about passing constants to these functions.

**procedure conv.u64Size( q:qword )**

Computes the output size of a 64-bit unsigned integer (passed in in q parameter) and returns this value in EAX. The return result will always be a value in the range 1..20 (e.g., "18446744073709551615") if the internal underscores flag contains false, 1..26 if the underscores flag contains true (e.g., "18\_446\_744\_073\_709\_551\_615").

HLA high-level calling sequence examples:

```
conv.u64Size( qwordVariable );
```

```

mov( eax, numSize );

conv.u64Size( <constant> );      // Must fit into 64 bits
mov( al, constantsSize );

```

HLA low-level calling sequence examples:

```

push( (type dword qwordVariable[4])); // Push H.O. dword first
push( (type dword qwordVariable[0])); // Push L.O. dword second
call conv.u64Size;
mov( eax, numSize );

```

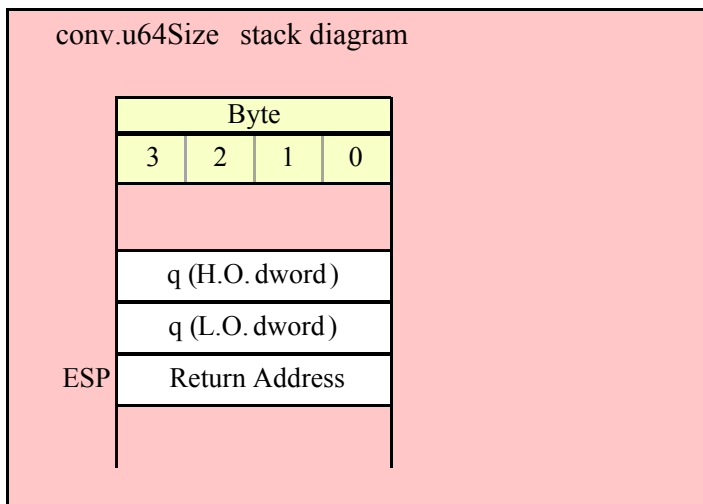
// Compute the size of a 64-bit constant:

```

pushd( 12345 >> 32 ); // Push H.O. dword first
pushd( 12345 & $FFFF_FFFF ); // Push L.O. dword second
call conv.u64Size;
mov( eax, constsSize );

```

See the comment at the end of `conv.u8Size` about passing constants to these functions. If you make a habit of explicitly passing 64-bit constants to this function, you might consider writing a macro to push the 64-bit constant for you (see the chapter on "Passing Parameters to Standard Library Routines" for more details).



#### **procedure conv.ul28Size( l:lword )**

Computes the output size of a 128-bit unsigned integer (passed in the `l` parameter) and returns this value in EAX. The return result will always be a value in the range 1..39 (e.g., "340282366920938463463374607431768211455") if the internal underscores flag contains false, 1..51 if the underscores flag contains true (e.g., "340\_282\_366\_920\_938\_463\_463\_374\_607\_431\_768\_211\_455").

HLA high-level calling sequence examples:

```

conv.ul28Size( lwordVariable );
mov( eax, numSize );

conv.ul28Size( <constant> );      // Must fit into 128 bits
mov( al, constantsSize );

```



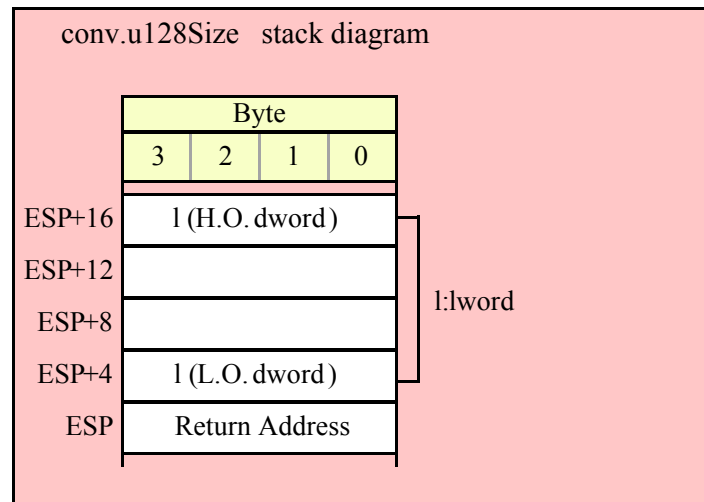
HLA low-level calling sequence examples:

```
push( (type dword lwordVariable[12])); // Push H.O. dword first
push( (type dword lwordVariable[8]));
push( (type dword lwordVariable[4]));
push( (type dword lwordVariable[0])); // Push L.O. dword last
call conv.u64Size;
mov( eax, numSize );
```

// Compute the size of a 128-bit constant:

```
pushd( 12345 >> 96 ); // Push H.O. dword first
pushd( (12345 >> 64) & $FFFF_FFFF );
pushd( (12345 >> 32) & $FFFF_FFFF );
pushd( 12345 & $FFFF_FFFF ); // Push L.O. dword last
call conv.u128Size;
mov( eax, constsSize );
```

See the comment at the end of `conv.u8Size` about passing constants to these functions. If you make a habit of explicitly passing 128-bit constants to this function, you might consider writing a macro to push the 128-bit constant for you (see the chapter on "Passing Parameters to Standard Library Routines" for more details).



### 8.5.3 Unsigned Integer Numeric to Buffer Conversions

These routines convert the input parameter to a sequence of characters and store those characters starting at location [EDI]. They return EDI pointing at the first character beyond the converted string. Note that these functions do not zero terminate the string; if you want a zero-terminated string, then store a zero at the byte pointed at by EDI upon return from these functions.

If the internal underscores flag is set (see `conv.getUnderscores` and `conv.setUnderscores` for details), then these functions will insert an underscore between each group of three digits starting with the least significant digit.

**procedure** `conv.u8ToBuf( u8: uns8 in al )`

This function converts the 8-bit unsigned integer passed in AL to a sequence of 1..3 characters. The string this function produces is always in the range 0..255. Note that because this string always contains three or fewer digits, the internal underscores flag setting does not affect this function's output.

HLA high-level calling sequence examples:

```
// The following will load "byteVariable" into AL and
// the address of "charArrayVariable" into EDI and then
// call conv.u8ToBuf:

conv.u8ToBuf( byteVariable, charArrayVariable );

// The following call will copy BH into AL and
// EDX into EDI prior to calling conv.u8ToBuf:

conv.u8ToBuf( bh, [edx] );

// The following just calls conv.u8ToBuf as AL and EDI
// already hold the parameter values:

conv.u8ToBuf( al, [edi] );

// The following loads the constant in AL and calls
// conv.u8ToBuf:

conv.u8ToBuf( <constant>, [edi] ); // <constant> must fit in 8 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the AL and EDI registers with their respective parameter values. In particular, you should not specify [EAX] as the buffer address because the code that HLA generates can overwrite the L.O. byte of EAX (i.e., AL) before it copies the address to the EDI register. It goes without saying that this function will overwrite the values of EAX and EDI if the original parameters are not AL and [EDI].

HLA low-level calling sequence examples:

```
// Passing a byte variable and a buffer variable:

mov( byteVariable, al );
lea( edi, charArrayVariable );
call conv.u8ToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

mov( byteVariable, al );
mov( &charArrayVariable, edi );
call conv.u8ToBuf;

// Passing a pair of registers (that are not
// AL and EDI):

mov( bh, al );
mov( edx, edi );
call conv.u8ToBuf;

// Passing a constant:

mov( <constant>, al );
call conv.u8ToBuf; // Assume EDI already contains buffer address.
```

**procedure conv.u16ToBuf( u16: uns16 in ax )**

This function converts the 16-bit unsigned integer passed in AX to a sequence of 1..5 characters if the internal underscores flag is false, 1..6 characters if the underscores flag contains true. The string this function produces is always in the range 0..65535. If the internal underscores flag contains true and the value is greater than 999, then this function emits an underscore between the third and fourth digits (from the right) in the string.

HLA high-level calling sequence examples:

```
// The following will load "wordVariable" into AX and
// the address of "charArrayVariable" into EDI and then
// call conv.u16ToBuf:

conv.u16ToBuf( wordVariable, charArrayVariable );

// The following call will copy BX into AX and
// EDX into EDI prior to calling conv.u16ToBuf:

conv.u16ToBuf( bx, [edx] );

// The following just calls conv.u16ToBuf as AX and EDI
// already hold the parameter values:

conv.u16ToBuf( ax, [edi] );

// The following loads the constant in AX and calls
// conv.u16ToBuf:

conv.u16ToBuf( <constant>, [edi] ); // <constant> must fit in 16 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the AX and EDI registers with their respective parameter values. In particular, you should not specify [EAX] as the buffer address because the code that HLA generates can overwrite the L.O. word of EAX (i.e., AX) before it copies the address to the EDI register. It goes without saying that this function will overwrite the values of EAX and EDI if the original parameters are not AX and [EDI].

HLA low-level calling sequence examples:

```
// Passing a word variable and a buffer variable:

mov( wordVariable, ax );
lea( edi, charArrayVariable );
call conv.u16ToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

mov( wordVariable, ax );
mov( &charArrayVariable, edi );
call conv.u16ToBuf;

// Passing a pair of registers (that are not
// AX and EDI):

mov( bx, ax );
mov( edx, edi );
call conv.u16ToBuf;

// Passing a constant:

mov( <constant>, ax );
```

```
call conv.u16ToBuf; // Assume EDI already contains buffer address.
```

#### **procedure conv.u32ToBuf( u32: uns32 in eax)**

This function converts the 32-bit unsigned integer passed in EAX to a sequence of 1..10 characters if the internal underscores flag is false, 1..11 characters if the underscores flag contains true. The string this function produces is always in the range 0..4294967295. If the internal underscores flag contains true and the value is greater than 999, then this function emits an underscore between the third and fourth digits (from the right) in the string.

HLA high-level calling sequence examples:

```
// The following will load "dwordVariable" into EAX and
// the address of "charArrayVariable" into EDI and then
// call conv.u32ToBuf:

conv.u32ToBuf( dwordVariable, charArrayVariable );

// The following call will copy EBX into EAX and
// EDX into EDI prior to calling conv.u32ToBuf:

conv.u32ToBuf( ebx, [edx] );

// The following just calls conv.u32ToBuf as EAX and EDI
// already hold the parameter values:

conv.u32ToBuf( eax, [edi] );

// The following loads the constant in EAX and calls
// conv.u32ToBuf:

conv.u32ToBuf( <constant>, [edi] ); // <constant> must fit in 32 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the EAX and EDI registers with their respective parameter values. In particular, you should not specify [EAX] as the buffer address because the code that HLA generates can overwrite EAX before it copies the address to the EDI register. It goes without saying that this function will overwrite the values of EAX and EDI if the original parameters are not EAX and [EDI].

HLA low-level calling sequence examples:

```
// Passing a dword variable and a buffer variable:

mov( dwordVariable, eax );
lea( edi, charArrayVariable );
call conv.u32ToBuf;

// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

mov( dwordVariable, eax );
mov( &charArrayVariable, edi );
call conv.u32ToBuf;

// Passing a pair of registers (that are not
// EAX and EDI):

mov( ebx, eax );
mov( edx, edi );
```

```

call conv.u32ToBuf;

// Passing a constant:

mov( <constant>, eax );
call conv.u32ToBuf; // Assume EDI already contains buffer address.

```

#### **procedure conv.u64ToBuf( q:qword )**

This function converts the 64-bit unsigned integer passed in q to a sequence of 1..20 characters if the internal underscores flag is false, 1..26 characters if the underscores flag contains true. The string this function produces is always in the range 0 .. 18446744073709551615. If the internal underscores flag contains true and the value is greater than 999, then this function emits an underscore between each group of three digits starting with the least significant digit.

HLA high-level calling sequence examples:

```

// The following will push the value of "qwordVariable"
// onto the stack, load the address of "charArrayVariable"
// into EDI and then call conv.u64ToBuf:

conv.u64ToBuf( qwordVariable, charArrayVariable );

// The following pushes the constant onto the stack and calls
// conv.u64ToBuf:

conv.u64ToBuf( <constant>, [edi] ); // <constant> must fit in 64 bits

```

When using the HLA high-level calling form, always keep in mind that these statements load the EDI register with the respective parameter value. It goes without saying that this function will overwrite the value of EDI if the original parameter is not [EDI].

HLA low-level calling sequence examples:

```

// Passing a qword variable and a buffer variable:

    push( (type dword qwordVariable[4])); // H.O. dword first
    push( (type dword qwordVariable[0])); // L.O. dword last
    lea( edi, charArrayVariable );
    call conv.u64ToBuf;

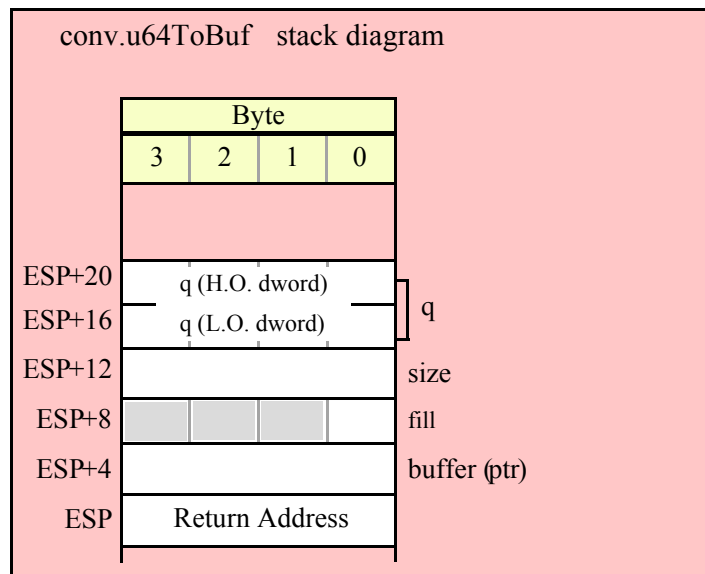
// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

    push( (type dword qwordVariable[4])); // H.O. dword first
    push( (type dword qwordVariable[0])); // L.O. dword last
    mov( &charArrayVariable, edi );
    call conv.u64ToBuf;

// Passing a constant:

    pushd( <constant> >> 32 ); // Push H.O. dword of constant first.
    pushd( <constant> & $FFFF_FFFF ); // Push L.O. dword second.
    call conv.u64ToBuf; // Assume EDI already contains buffer address.

```



#### procedure conv.u128ToBuf( l:lword )

This function converts the 128-bit unsigned integer passed in *l* to a sequence of 1..39 characters if the internal underscores flag is false, 1..52 characters if the underscores flag contains true. The string this function produces is always in the range 0 .. 340282366920938463463374607431768211455. If the internal underscores flag contains true and the value is greater than 999, then this function emits an underscore between each group of three digits starting with the least significant digit.

HLA high-level calling sequence examples:

```
// The following will push the value of "lwordVariable"
// onto the stack, load the address of "charArrayVariable"
// into EDI and then call conv.u128ToBuf:
```

```
conv.u128ToBuf( lwordVariable, charArrayVariable );
```

```
// The following pushes the constant onto the stack and calls
// conv.u128ToBuf:
```

```
conv.u128ToBuf( <constant>, [edi] ); // <constant> must fit in 128 bits
```

When using the HLA high-level calling form, always keep in mind that these statements load the EDI register with the respective parameter value. It goes without saying that this function will overwrite the value of EDI if the original parameter is not [EDI].

HLA low-level calling sequence examples:

```
// Passing an lword variable and a buffer variable:
```

```
push( (type dword lwordVariable[12])); // H.O. dword first
push( (type dword lwordVariable[8]));
push( (type dword lwordVariable[4]));
push( (type dword lwordVariable[0])); // L.O. dword last
lea( edi, charArrayVariable );
```

```

call conv.u128ToBuf;

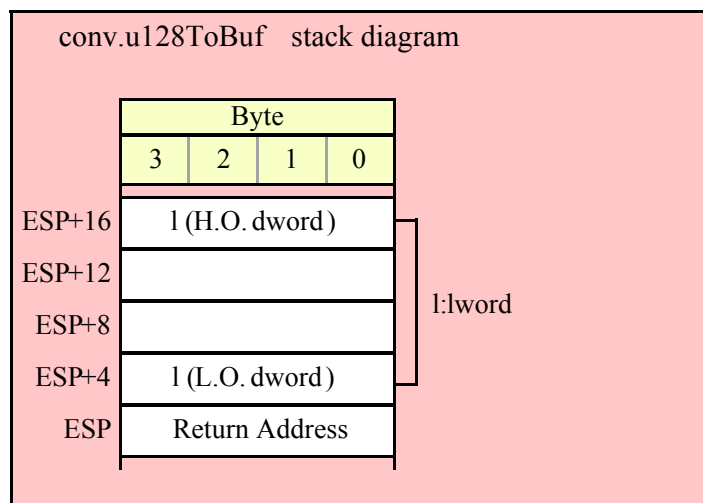
// Alternate form of above if charArrayVariable is
// a static object (STATIC, READONLY, or STORAGE):

    push( (type dword lwordVariable[12])); // H.O. dword first
    push( (type dword lwordVariable[8]));
    push( (type dword lwordVariable[4]));
    push( (type dword lwordVariable[0])); // L.O. dword last
mov( &charArrayVariable, edi );
call conv.u128ToBuf;

// Passing a constant:

pushd( <constant> >> 96 ); // Push H.O. dword of constant first.
pushd( (<constant> >> 64) & $FFFF_FFFF );
pushd( (<constant> >> 32) & $FFFF_FFFF );
pushd( <constant> & $FFFF_FFFF ); // Push L.O. dword last.
call conv.u128ToBuf; // Assume EDI already contains buffer address.

```



## 8.5.4 Unsigned Integer Numeric to String Conversions

These routines convert an unsigned integer value ( 8, 16, 32, 64, or 128 bits) to a string. The standard ("unadorned") functions store the string data into a string object that you pass as a parameter to the function. That string object must be preallocated and large enough to receive the string result (else a string overflow occurs). The "adorned" functions, whose names begin with "a\_" automatically allocate storage on the heap, store the converted string into that heap object, and then return a pointer to the newly allocated string in the EAX register (it is the caller's responsibility to free the storage when it is no longer needed).

These functions let you specify a minimum field width and a fill character. If the number would require fewer than width print positions, the routines copy the fill character to the remaining positions in the destination string. If width is positive, the number is right justified in the string. If width is negative, the number is left justified in the string. If the string representation of the value requires more than width print positions, then these functions ignore the width and fill parameters and use however many positions are necessary to properly display the value.

```
xxxToStr ( value, width, fill, buffer );
```

Assuming “value” requires five print positions, “width” is eight, and fill is “f” then the xxxToStr functions produce the string

f	f	f	V	A	L	U	E
---	---	---	---	---	---	---	---

Assuming “value” requires five print positions, “width” is minus eight, and fill is “f” then the xxxToStr functions produce the string

V	A	L	U	E	f	f	f
---	---	---	---	---	---	---	---

Here are the maximum number of print positions these routines will produce for each data type before considering the minimum field width:

Underscores flag is false:

```
8 bits:3 (0..255)
16 bits:5 (0..65535)
32 bits:10 (0..4294967295)
64 bits:20 (0..18446744073709551615)
128 bits:39 (0 .. 340282366920938463463374607431768211455)
```

Underscores flag is true:

```
8 bits:3 (0..255)
16 bits:6 (0..65_535)
32 bits:13 (0..4_294_967_295)
64 bits:26 (0..18_446_744_073_709_551_615)
128 bits:51 (0..340_282_366_920_938_463_463_374_607_431_768_211_455)
```

```
procedure conv.u8ToStr ( b:uns8; width:int32; fill:char; buffer:string );
```

This function converts an 8-bit unsigned integer to the decimal string representation of that integer and stores the string in the preallocated string object specified by the dest parameter. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). This function will raise a string overflow exception if the destination string is not large enough to hold the conversion. Note that the internal underscores flag will not affect the output because 8-bit integers are always three digits or smaller.

HLA high-level calling sequence examples:

```
// The following will push "byteVariable" and "destStr" (which
// is a pointer to a string object) and then call conv.u8ToStr:

conv.u8ToStr( byteVariable, destStr );

// The following call will BH's value onto the stack and then
// push EDX's value (which is the address of a string object)
// before calling conv.u8ToStr:

conv.u8ToStr( bh, edx );

// The following pushes the constant and destStr and calls
// conv.u8ToStr:

conv.bToBuf( <constant>, destStr ); // <constant> must fit in 8 bits
```

When using the HLA high-level calling form, remember that string variables are dword pointers that contain the address of a string object. The destination string parameter is passed by value, not by reference; it just turns



out that the value of a string is a pointer to the actual string data (confusing, isn't it?). In any case, this is why you can pass a register value as the destination string location rather than having to pass something like "[edx]". A construct like "[edx]" would imply that EDX contains the address of the string variable, that is, a pointer to the pointer to the string.

HLA low-level calling sequence examples:

```
// Passing a byte variable and a buffer variable, option 1
// (if a 32-bit register is available):

movzx( byteVariable, eax );
push( eax );
push( destStr );
call conv.u8ToStr;

// Passing a byte variable and a buffer variable, option 2
// (if byteVariable isn't the last byte in mapped memory):

push( (type dword byteVariable));
push( destStr );
call conv.u8ToStr;

// Passing a byte variable and a buffer variable, option 3
// No registers are available and we can't guarantee that
// the three bytes following byteVariable are present in
// mapped memory:

sub( 4, esp );
push( eax );
movzx( byteVariable, eax );
mov( eax, [esp+4] );
pop( eax );
push( destStr );
call conv.u8ToStr;

// Passing a pair of registers (hex value in AL, BL, CL, or DL):
// BL = value to print, EDX = pointer to string object.

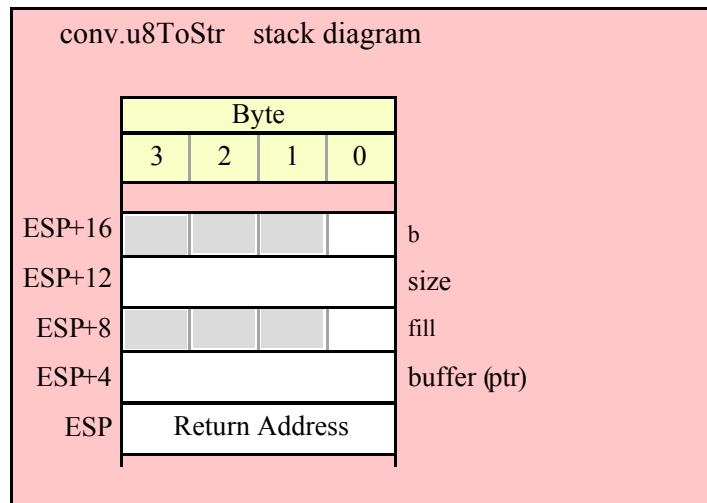
push( ebx );          // Pushes BL
push( edx );
call conv.u8ToStr;

// Passing a pair of registers (hex value in AH, BH, CH, or DH):
// BH = value to print, EDX = pointer to string object.

pushd( 0 );
mov( bh, [esp] );
push( edx );
call conv.u8ToStr;

// Passing a constant:

pushd( <constant> );
push( destStr );
call conv.u8ToStr;
```



```
procedure conv.a_u8ToStr ( b:uns8; width:int32; fill:char );
    @returns( "eax" );
```

This function converts an 8-bit unsigned integer to the decimal string representation of that integer and stores the string in storage it allocates on the heap. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). The caller is responsible for freeing the storage when it is no longer needed. Note that the internal underscores flag will not affect the output because 8-bit integers are always three digits or smaller.

HLA high-level calling sequence examples:

```
// The following will push "byteVariable" and then call conv.a_u8ToStr:
```

```
conv.a_u8ToStr( byteVariable );
mov( eax, byteStr );
```

```
// The following call will BH's value onto the stack
// before calling conv.a_u8ToStr:
```

```
conv.a_u8ToStr( bh );
mov( eax, byteStr );
```

```
// The following pushes the constant and calls
// conv.a_u8ToStr:
```

```
conv.a_u8ToStr( <constant> ); // <constant> must fit in 8 bits
mov( eax, byteStr );
```

HLA low-level calling sequence examples:

```
// Passing a byte variable and a buffer variable, option 1
// (if a 32-bit register is available):
```

```
movzx( byteVariable, eax );
push( eax );
```

```

call conv.a_u8ToStr;
mov( eax, destStr );

// Passing a byte variable and a buffer variable, option 2
// (if byteVariable isn't the last byte in mapped memory):

push( (type dword byteVariable));
call conv.a_u8ToStr;
mov( eax, destStr );

// Passing a byte variable, option 3
// No registers are available and we can't guarantee that
// the three bytes following byteVariable are present in
// mapped memory:

sub( 4, esp );
push( eax );
movzx( byteVariable, eax );
mov( eax, [esp+4] );
pop( eax );
call conv.a_u8ToStr;
mov( eax, destStr );

// Passing a pair of registers (hex value in AL, BL, CL, or DL):
// BL = value to print, EDX = pointer to string object.

push( ebx );          // Pushes BL
call conv.a_u8ToStr;
mov( eax, byteStr );

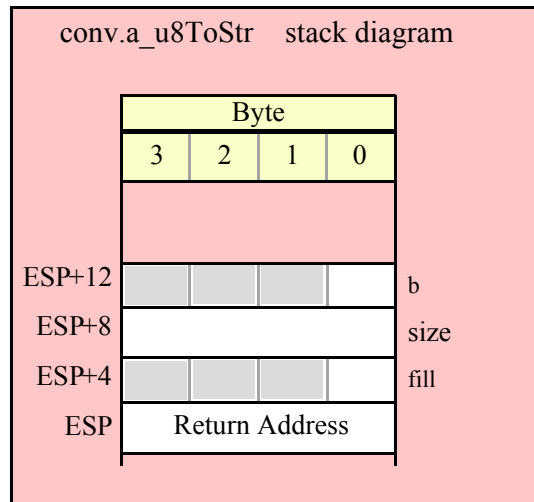
// Passing a pair of registers (hex value in AH, BH, CH, or DH):
// BH = value to print, EDX = pointer to string object.

pushd( 0 );
mov( bh, [esp] );
call conv.a_u8ToStr;
mov( eax, byteStr );

// Passing a constant:

pushd( <constant> );
call conv.a_u8ToStr;
mov( eax, byteStr );

```



```
procedure conv.ul6ToStr( w:uns16; width:int32; fill:char; buffer:string );
```

This function converts a 16-bit unsigned integer to its decimal string representation and stores the string in the preallocated string object specified by the dest parameter. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). This function will raise a string overflow exception if the destination string is not large enough to hold the conversion. If the conversion requires more than three digits and the internal underscores flag is true, then this function will insert an underscore between each group of three digits, starting with the least significant digit (see conv.getUnderscores and conv.setUnderscores for more details).

HLA high-level calling sequence examples:

```
// The following will push "wordVariable" and "destStr" (which
// is a pointer to a string object) and then call conv.ul6ToStr:

conv.ul6ToStr( wordVariable, destStr );

// The following call will BX's value onto the stack and then
// push EDX's value (which is the address of a string object)
// before calling conv.ul6ToStr:

conv.ul6ToStr( bx, edx );

// The following pushes the constant and destStr and calls
// conv.ul6ToStr:

conv.ul6ToStr( <constant>, destStr ); // <constant> must fit in 16 bits
```

When using the HLA high-level calling form, remember that string variables are dword pointers that contain the address of a string object. The destination string parameter is passed by value, not by reference; it just turns out that the value of a string is a pointer to the actual string data (confusing, isn't it?). In any case, this is why you can pass a register value as the destination string location rather than having to pass something like "[edx]". A construct like "[edx]" would imply that EDX contains the address of the string variable, that is, a pointer to the pointer to the string.

HLA low-level calling sequence examples:

```
// Passing a word variable and a buffer variable, option 1
```

```
// (if a 32-bit register is available):

movzx( byteVariable, eax );
push( eax );
push( destStr );
call conv.u16ToStr;

// Passing a word variable and a buffer variable, option 2
// (if wordVariable isn't the last byte in mapped memory):

push( (type dword wordVariable));
push( destStr );
call conv.u16ToStr;

// Passing a word variable and a buffer variable, option 3
// No registers are available and we can't guarantee that
// the two bytes following wordVariable are present in
// mapped memory:

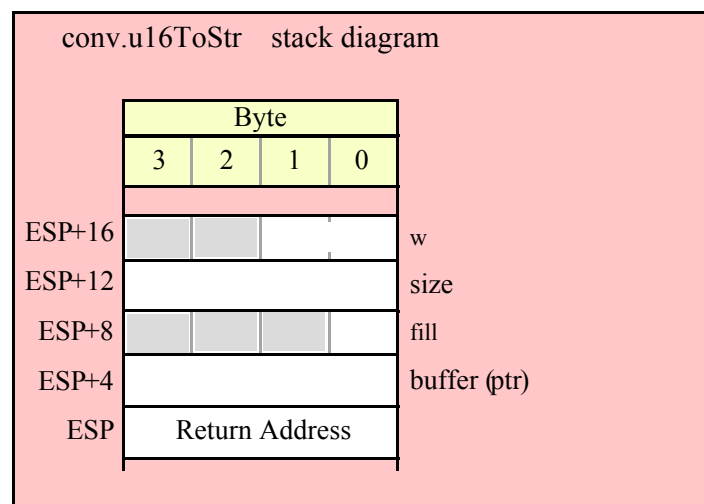
pushw( 0 );
push( wordVariable );
push( destStr );
call conv.u16ToStr;

// Passing a pair of registers:
// BX = value to print, EDX = pointer to string object.

push( ebx );           // Pushes BX
push( edx );
call conv.u16ToStr;

// Passing a constant:

pushd( <constant> );
push( destStr );
call conv.u16ToStr;
```



```

procedure conv.a_u16ToStr( w:uns16; width:int32; fill:char );
    @returns( "eax" );

```

This function converts a 16-bit unsigned integer to the decimal string representation of that integer and stores the string in storage it allocates on the heap. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). The caller is responsible for freeing the storage when it is no longer needed. If the conversion produces more than three digits and the internal underscores flag is true, then this function will insert an underscore between each group of three digits, starting with the least significant digit (see conv.getUnderscores and conv.setUnderscores for more details).

HLA high-level calling sequence examples:

```

// The following will push "wordVariable" and call conv.a_u16ToStr:

conv.a_u16ToStr( wordVariable );
mov( eax, destStr );

// The following call will BX's value onto the stack
// before calling conv.a_u16ToStr:

conv.a_u16ToStr( bx );

// The following pushes the constant and calls
// conv.a_u16ToStr:

conv.a_u16ToStr( <const>, destStr ); // <const> must fit in 16 bits

```

HLA low-level calling sequence examples:

```

// Passing a word variable and a buffer variable, option 1
// (if a 32-bit register is available):

movzx( wordVariable, eax );
push( eax );
call conv.a_u16ToStr;
mov( eax, destStr );

// Passing a word variable and a buffer variable, option 2
// (if wordVariable isn't the last byte in mapped memory):

push( (type dword wordVariable));
call conv.a_u16ToStr;
mov( eax, destStr );

// Passing a word variable and a buffer variable, option 3
// No registers are available and we can't guarantee that
// the two bytes following wordVariable are present in
// mapped memory:

pushw( 0 );
push( wordVariable );
call conv.a_u16ToStr;
mov( eax, destStr );

// Passing a pair of registers:
// BX = value to print.

```

```

push( ebx );          // Pushes BX
call conv.a_u16ToStr;
mov( eax, wordStr );

```

```

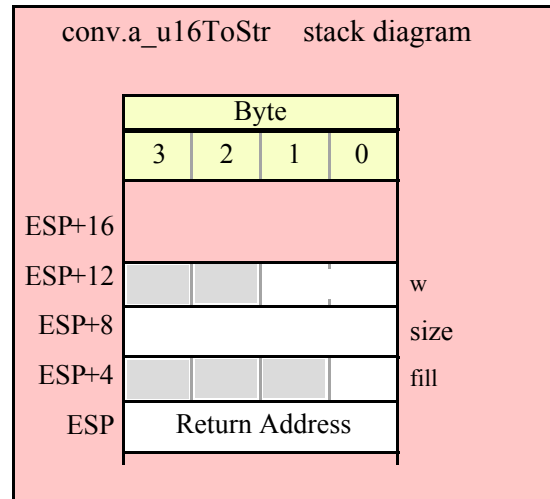
// Passing a constant:

```

```

pushd( <constant> );
call conv.a_u16ToStr;
mov( eax, destStr );

```



```

procedure conv.u32ToStr( d:uns32; width:int32; fill:char; buffer:string );

```

This function converts a 32-bit unsigned integer to its decimal string representation and stores the string in the preallocated string object specified by the `buffer` parameter. The `width` and `fill` parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). This function will raise a string overflow exception if the destination string is not large enough to hold the conversion. If the conversion requires more than three digits and the internal `underscores` flag is true, then this function will insert an underscore between each group of three digits, starting with the least significant digit (see `conv.getUnderscores` and `conv.setUnderscores` for more details).

HLA high-level calling sequence examples:

```

// The following will push "dwordVariable" and "destStr" (which
// is a pointer to a string object) and then call conv.u32ToStr:

```

```

conv.u32ToStr( dwordVariable, destStr );

```

```

// The following call will push EBX's value onto the stack and then
// push EDX's value (which is the address of a string object)
// before calling conv.u32ToStr:

```

```

conv.u32ToStr( ebx, edx );

```

```

// The following pushes the constant and destStr and calls
// conv.u32ToStr:

```

```
conv.u32ToStr( <constant>, destStr ); // <constant> must fit in 32 bits
```

When using the HLA high-level calling form, remember that string variables are dword pointers that contain the address of a string object. The destination string parameter is passed by value, not by reference; it just turns out that the value of a string is a pointer to the actual string data. In any case, this is why you can pass a register value as the destination string location rather than having to pass something like "[edx]". A construct like "[edx]" would imply that EDX contains the address of the string variable, that is, a pointer to the pointer to the string data.

HLA low-level calling sequence examples:

```
// Passing a dword variable and a buffer variable:
```

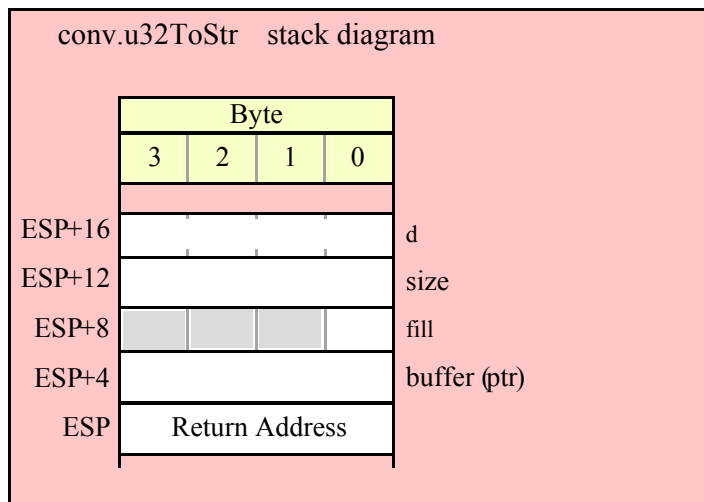
```
push( dwordVariable );
push( destStr );
call conv.u32ToStr;
```

```
// Passing a pair of registers:
// EBX = value to print, EDX = pointer to string object.
```

```
push( ebx );
push( edx );
call conv.u32ToStr;
```

```
// Passing a constant:
```

```
pushd( <constant> );
push( destStr );
call conv.u32ToStr;
```





```
procedure conv.a_u32ToStr( d:uns32; width:int32; fill:char );
    @returns( "eax" );
```

This function converts a 32-bit unsigned integer to the decimal string representation of that integer and stores the string in storage it allocates on the heap. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). The caller is responsible for freeing the storage when it is no longer needed. If the conversion produces more than three digits and the internal underscores flag is true, then this function will insert an underscore between each group of three digits, starting with the least significant digit (see conv.getUnderscores and conv.setUnderscores for more details).

HLA high-level calling sequence examples:

```
// The following will push "dwordVariable" and then call conv.a_u32ToStr:
```

```
conv.a_u32ToStr( dwordVariable );
mov( eax, destStr );
```

```
// The following call will push EBX's value onto the stack
// before calling conv.a_u32ToStr:
```

```
conv.a_u32ToStr( ebx );
```

```
// The following pushes the constant and calls
// conv.a_u32ToStr:
```

```
conv.a_u32ToStr( <constant>, destStr ); // <constant> must fit in 32 bits
```

HLA low-level calling sequence examples:

```
// Passing a dword variable:
```

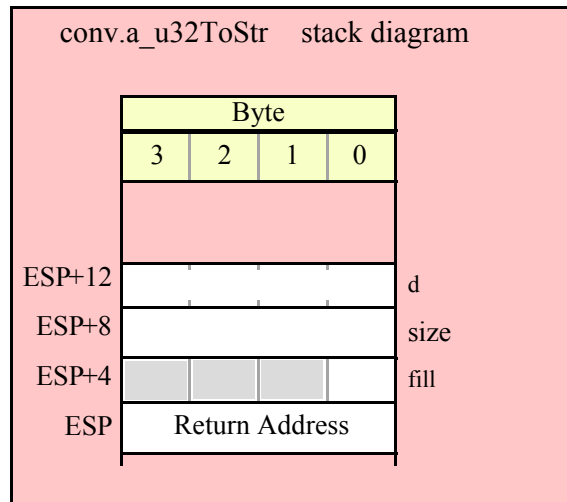
```
push( dwordVariable );
call conv.a_u32ToStr;
mov( eax, destStr );
```

```
// Passing a register:
// EBX = value to print.
```

```
push( ebx );
call conv.a_u32ToStr;
mov( eax, dwordStr );
```

```
// Passing a constant:
```

```
pushd( <constant> );
call conv.a_u32ToStr;
mov( eax, destStr );
```



```
procedure conv.u64ToStr( q:qword; width:int32; fill:char; buffer:string );
```

This function converts a 64-bit unsigned integer to its decimal string representation and stores the string in the preallocated string object specified by the buffer parameter. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). This function will raise a string overflow exception if the destination string is not large enough to hold the conversion. If the conversion requires more than three digits and the internal underscores flag is true, then this function will insert an underscore between each group of three digits, starting with the least significant digit (see conv.getUnderscores and conv.setUnderscores for more details).

HLA high-level calling sequence examples:

```
// The following will push the value of "qwordVariable"
// and the value of the destStr string variable
// onto the stack and then call conv.u64ToStr:
```

```
conv.u64ToStr( qwordVariable, destStr );
```

```
// The following pushes the constant onto the stack along with
// the value held in the destStr variable and calls
// conv.u64ToStr:
```

```
conv.u64ToStr( <constant>, destStr ); // <constant> must fit in 64 bits
```

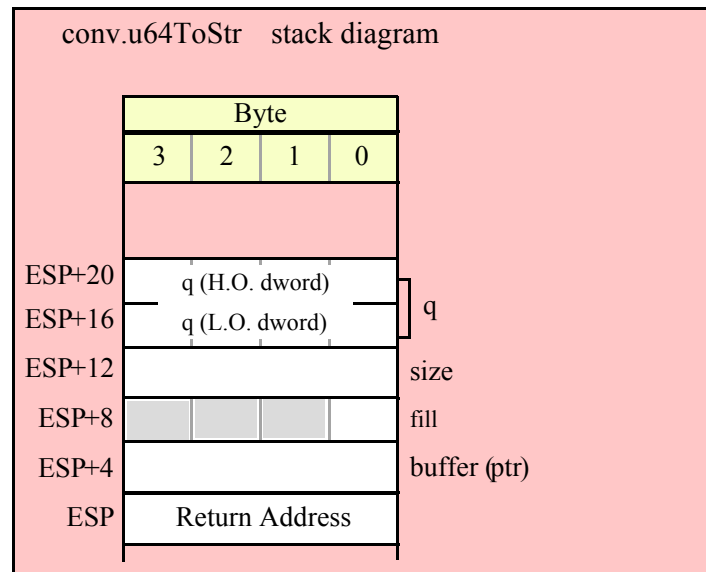
HLA low-level calling sequence examples:

```
// Passing a qword variable and a buffer variable:
```

```
    push( (type dword qwordVariable[4])); // H.O. dword first
    push( (type dword qwordVariable[0])); // L.O. dword last
    push( destStr );
    call conv.u64ToStr;
```

```
// Passing a constant:

pushd( <constant> >> 32 );// Push H.O. dword of constant first.
pushd( <constant> & $FFFF_FFFF );// Push L.O. dword second.
push( destStr );
call conv.u64ToStr;
```



```
procedure conv.a_u64ToStr( q:qword; width:int32; fill:char );
    @returns( "eax" );
```

This function converts a 64-bit unsigned integer to the decimal string representation of that integer and stores the string in storage it allocates on the heap. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). The caller is responsible for freeing the storage when it is no longer needed. If the conversion produces more than three digits and the internal underscores flag is true, then this function will insert an underscore between each group of three digits, starting with the least significant digit (see `conv.getUnderscores` and `conv.setUnderscores` for more details).

HLA high-level calling sequence examples:

```
// The following will push the value of "qwordVariable"
// onto the stack and then call conv.a u64ToStr:
```

```
conv.a_u64ToStr( qwordVariable );  
mov( eax, destStr );
```

```
// The following pushes the constant onto the stack and calls
// conv.a u64ToStr:
```

```
conv.a_u64ToStr( <constant> ); // <constant> must fit in 64 bits
mov( eax, destStr );
```

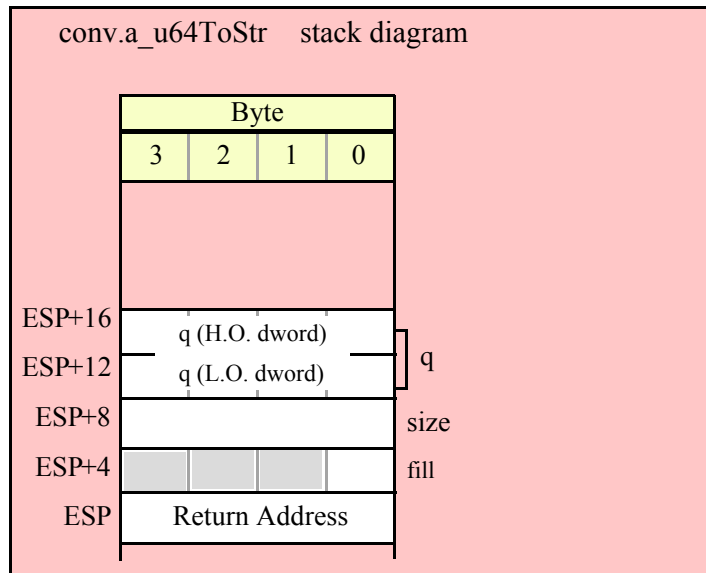
HLA low-level calling sequence examples:

```
// Passing a qword variable:
```

```
    push( (type dword qwordVariable[4])); // H.O. dword first
    push( (type dword qwordVariable[0])); // L.O. dword last
    call conv.a_u64ToStr;
    mov( eax, destStr );
```

```
// Passing a constant:
```

```
    pushd( <constant> >> 32 ); // Push H.O. dword of constant first.
    pushd( <constant> & $FFFF_FFFF ); // Push L.O. dword second.
    call conv.a_u64ToStr;
    mov( eax, destStr );
```



```
procedure conv.u128ToStr(l:lword; width:int32; fill:char; buffer:string );
```

This function converts a 128-bit unsigned integer to its decimal string representation and stores the string in the preallocated string object specified by the buffer parameter. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). This function will raise a string overflow exception if the destination string is not large enough to hold the conversion. If the conversion requires more than three digits and the internal underscores flag is true, then this function will insert an underscore between each group of three digits, starting with the least significant digit (see conv.getUnderscores and conv.setUnderscores for more details).

HLA high-level calling sequence examples:

```
// The following will push the value of "lwordVariable"
// and destStr onto the stack, and then call conv.u128ToStr:
```

```
conv.u128ToStr( lwordVariable, destStr );
```

```
// The following pushes the constant onto the stack and calls
// conv.u128ToStr:
```

```
conv.ul28ToStr( <constant>, edx ); // EDX contains string pointer value.
```

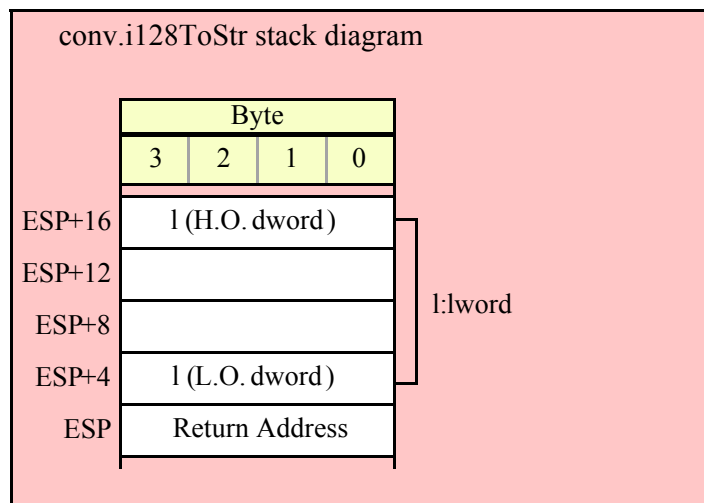
HLA low-level calling sequence examples:

```
// Passing an lword variable:
```

```
    push( (type dword lwordVariable[12])); // H.O. dword first
    push( (type dword lwordVariable[8]));
    push( (type dword lwordVariable[4]));
    push( (type dword lwordVariable[0])); // L.O. dword last
push( destStr );
call conv.ul28ToStr;
```

```
// Passing a constant:
```

```
pushd( <constant> >> 96 ); // Push H.O. dword of constant first.
pushd( (<constant> >> 64) & $FFFF_FFFF );
pushd( (<constant> >> 32) & $FFFF_FFFF );
pushd( <constant> & $FFFF_FFFF ); // Push L.O. dword last.
push( edx ); // EDX contains string pointer value.
call conv.ul28ToStr;
```



```
procedure conv.a_ul28ToStr( l:lword; width:int32; fill:char );
    @returns( "eax" );
```

This function converts a 128-bit signed integer to the decimal string representation of that integer and stores the string in storage it allocates on the heap. The width and fill parameters specify the minimum field width and padding character (if the minimum field width is greater than the number of output characters needed for the string). The caller is responsible for freeing the storage when it is no longer needed. If the conversion produces more than three digits and the internal underscores flag is true, then this function will insert an underscore between each group of three digits, starting with the least significant digit (see `conv.getUnderscores` and `conv.setUnderscores` for more details).

HLA high-level calling sequence examples:

```
// The following will push the value of "lwordVariable"
// onto the stack, and then call conv.a_ul28ToStr:
```

```
conv.a_u128ToStr( lwordVariable );
mov( eax, destStr );
```

```
// The following pushes the constant onto the stack and calls
// conv.a_u128ToStr:
```

```
conv.a_u128ToStr( <constant> );
mov( eax, destStr );
```

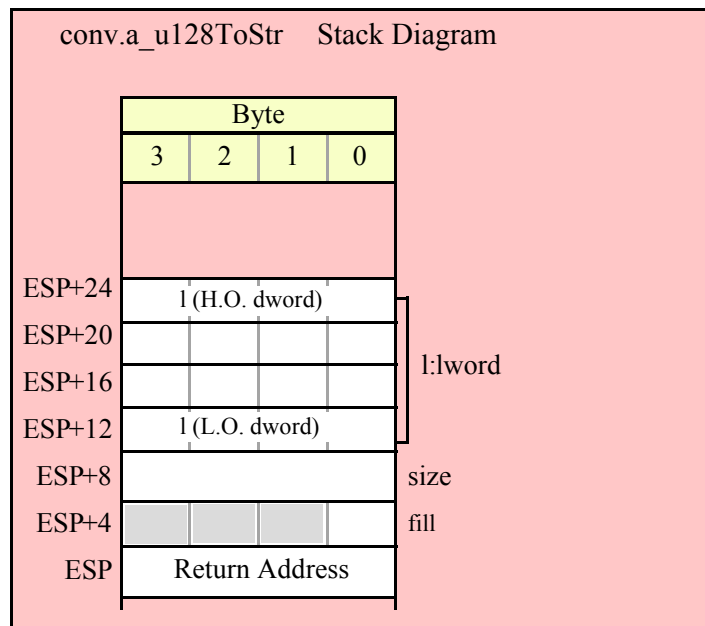
HLA low-level calling sequence examples:

```
// Passing an lword variable:
```

```
    push( (type dword lwordVariable[12])); // H.O. dword first
    push( (type dword lwordVariable[8]));
    push( (type dword lwordVariable[4]));
    push( (type dword lwordVariable[0])); // L.O. dword last
call conv.a_u128ToStr;
mov( eax, destStr );
```

```
// Passing a constant:
```

```
pushd( <constant> >> 96 ); // Push H.O. dword of constant first.
pushd( (<constant> >> 64) & $FFFF_FFFF );
pushd( (<constant> >> 32) & $FFFF_FFFF );
pushd( <constant> & $FFFF_FFFF ); // Push L.O. dword last.
call conv.a_u128ToStr;
mov( eax, destStr );
```



## 8.5.5 Unsigned Integer String to Numeric Conversions

The standard library string to integer conversion routines convert a sequence of digits into the corresponding unsigned integer value. These routines begin by skipping over any leading delimiter characters (see the `conv.getDelimiters` and `conv.setDelimiters` functions for details) followed by any number of decimal digits and underscores (these routines ignore the underscores). Conversion stops at the end of the string or upon encountering a delimiter character.

These routines will raise a conversion error exception if they encounter a 7-bit ASCII character that is not a decimal digit, an underscore, or a delimiter character during the translation. These routines will raise an illegal character exception if they encounter a non-ASCII character (one with its H.O. bit set). These routines will raise a value out of range exception if the converted value will not fit in the destination data object.

There are two basic sets of string to numeric conversion routines: the `conv.atou*` routines and the `conv.strTou*` routines. The `atou*` routines process the characters pointed at by the ESI register. The `strTou*` routines process data in a string object, starting at an offset specified by a second parameter. For example, `"conv.strTou8( "12345", 3);"` returns the value 45 because it begins processing the string at (zero-based) offset 3 in the string.

```
procedure conv.atou8 ( buffer: var in esi ); @returns( "ax" );
```

This function converts the sequence of characters starting at the memory address held in ESI to an 8-bit unsigned integer. It returns the value (in the range 0..+255) in AL. Note that this function actually returns the zero-extended value in EAX, so you may use EAX if it is more convenient to do so. Upon successful return, ESI is pointing at the delimiter character at the end of the sequence of digits.

HLA high-level calling sequence examples:

```
// The following will convert the decimal characters in memory
// at the address specified by [esi] into a numeric value
// and return that value in AL:
```

```
conv.atou8( [esi] );
mov( al, numericResult );
```

```
// The following loads ESI with the address of
// a sequence of decimal characters (held in an HLA
// string) and converts them to an 8-bit number:
```

```
conv.atou8( sourceStr ); // Loads "sourceStr" into ESI
mov( al, byteVariable );
```

HLA low-level calling sequence examples:

```
// Same as first example above - ESI already contains
// the address of the first character to convert:
```

```
call conv.atou8;
mov( al, numericResult );
```

```
// Same as second example above
```

```
static
  sourceStr :string := "12";
  .
  .
  .
mov( sourceStr, esi );
call conv.atou8;
mov( al, num12 );
```

```
procedure conv.atoul6 ( buffer: var in esi ); @returns( "ax" );
```

This function converts the sequence of characters starting at the memory address held in ESI to a 16-bit unsigned integer. It returns the value (in the range 0..65535) in AX. Note that this function actually returns the zero-extended value in EAX, so you may use EAX if it is more convenient to do so. Upon successful return, ESI is pointing at the delimiter character at the end of the sequence of digits.

HLA high-level calling sequence examples:

```
// The following will convert the decimal characters in memory
// at the address specified by [esi] into a numeric value
// and return that value in AX:
```

```
conv.atoul6( [esi] );
mov( ax, numericResult );
```

```
// The following loads ESI with the address of
// a sequence of decimal characters (held in an HLA
// string) and converts them to a 16-bit number:
```

```
conv.atoul6( sourceStr ); // Loads "sourceStr" into ESI
mov( ax, wordVariable );
```

HLA low-level calling sequence examples:

```
// Same as first example above - ESI already contains
// the address of the first character to convert:
```

```
call conv.atoul6;
mov( ax, numericResult );
```

```
// Same as second example above
```

```
static
  sourceStr :string := "12";
.
.
.
mov( sourceStr, esi );
call conv.atoul6;
mov( ax, num12 );
```

```
procedure conv.atou32 ( buffer: var in esi ); @returns( "eax" );
```

This function converts the sequence of characters starting at the memory address held in ESI to a 32-bit unsigned integer. It returns the value (in the range 0..4294967295) in EAX. ESI is pointing at the delimiter character at the end of the sequence of digits.

HLA high-level calling sequence examples:

```
// The following will convert the hexadecimal characters in memory
// at the address specified by [esi] into a numeric value
// and return that value in EAX:
```



```

conv.atou32( [esi] );
mov( eax, numericResult );

// The following loads ESI with the address of
// a sequence of decimal characters (held in an HLA
// string) and converts them to a 32-bit number:

conv.atou32( sourceStr ); // Loads "sourceStr" into ESI
mov( eax, dwordVariable );

```

HLA low-level calling sequence examples:

```

// Same as first example above - ESI already contains
// the address of the first character to convert:

call conv.atou32;
mov( eax, numericResult );

// Same as second example above

static
  sourceStr:string := "12";
.
.
.
mov( sourceStr, esi );
call conv.atou32;
mov( eax, num12 );

```

**procedure conv.atou64 ( buffer: var in esi ); @returns( "edx:eax" );**

This function converts the sequence of characters starting at the memory address held in ESI to a 64-bit unsigned integer. It returns the value (in the range 0..18446744073709551615) in EDX:EAX (EDX contains the H.O. dword). ESI is pointing at the delimiter character at the end of the sequence of digits.

HLA high-level calling sequence examples:

```

// The following will convert the decimal characters in memory
// at the address specified by [esi] into a numeric value
// and return that value in EDX:EAX:

conv.atou64( [esi] );
mov( eax, (type dword hex64NumericResult[0]) );
mov( edx, (type dword hex64NumericResult[4]) );

// The following loads ESI with the address of
// a sequence of decimal characters (held in an HLA
// string) and converts them to a 64-bit number:

conv.atou64( sourceStr ); // Loads "sourceStr" into ESI
mov( eax, (type dword qwordVariable[0]) );
mov( edx, (type dword qwordVariable[4]) );

```

HLA low-level calling sequence examples:

```
// Same as first example above - ESI already contains
// the address of the first character to convert:

    call conv.atou64;
mov( eax, (type dword numericResult[0]) );
mov( edx, (type dword numericResult[4]) );

// Same as second example above

static
    sourceStr:string := "12";
.
.
.
mov( sourceStr, esi );
call conv.atou64;
mov( eax, (type dword qwordVariable[0]) );
mov( edx, (type dword qwordVariable[4]) );
```

**procedure conv.atoul28( buffer: var in esi; var l:word );**

This function converts the sequence of characters starting at the memory address held in ESI to a 128-bit signed integer. It returns the value (in the range 0..340282366920938463463374607431768211455) in the l parameter that is passed by reference to this function. ESI is pointing at the delimiter character at the end of the sequence of digits.

HLA high-level calling sequence examples:

```
// The following will convert the decimal characters in memory
// at the address specified by [esi] into a numeric value
// and stores that value in lwordDest (passed by reference):

conv.atoul28( [esi], lwordDest );

// The following loads ESI with the address of
// a sequence of decimal characters (held in an HLA
// string) and converts them to a 128-bit number that it
// stores in lwordDest:

conv.atoul28( sourceStr, lwordDest ); // Loads "sourceStr" into ESI
```

HLA low-level calling sequence examples:

```
// Option 1: lwordDest is a static object declared in a
// HLA STATIC, READONLY, or STORAGE section:
// As with the first example above, assume ESI already
// contains the address of the string to convert:

pushd( &lwordDest ); // Pass address of lwordDest as reference parm.
call conv.atoul28;

// Option 2: lwordDest is a simple automatic variable (no indexing)
```

```

// declared in a VAR section (or as a parameter). Assume that
// no 32-bit registers can be disturbed by this code.
// As with the first example above, assume ESI already
// contains the address of the string to convert:

push( ebp );
add( @offset( lwordDest ), (type dword [esp]));
call conv.atoul28;

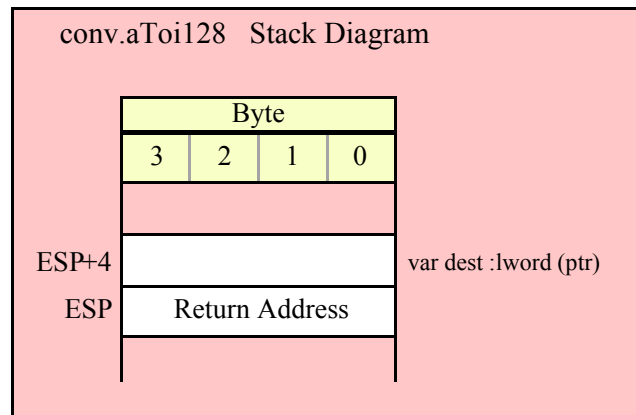
// Option 3: lwordDest is a complex addressing mode and at least
// one 32-bit register is available for use by this code.
// As with the first example above, assume ESI already
// contains the address of the string to convert:

lea( eax, lwordDest );// Assume EAX is the available register
push( eax );
call conv.atoul28;

// Same as second high-level example above. Assumes that
// lwordDest is a static object.

static
  sourceStr :string := "12";
  .
  .
  .
mov( sourceStr, esi );
pushd( &lwordDest );
call conv.atoul28;

```



```
procedure conv.strTou8( s:string; index:dword )
```

This function converts the sequence of characters starting at zero-based offset index within the string parameter *s* to an 8-bit unsigned integer. It returns the value (in the range 0..255) in AL. Note that this function actually returns the zero-extended value in EAX, so you may use EAX if it is more convenient to do so.

HLA high-level calling sequence examples:

```

// The following will convert the characters at the beginning
// of "decValueStr" to numeric form:

```

```
conv.strTou8( decValueStr, 0 );// Index=0 starts at beginning
mov( al, numericResult );
```

// The following demonstrates using a non-zero index:

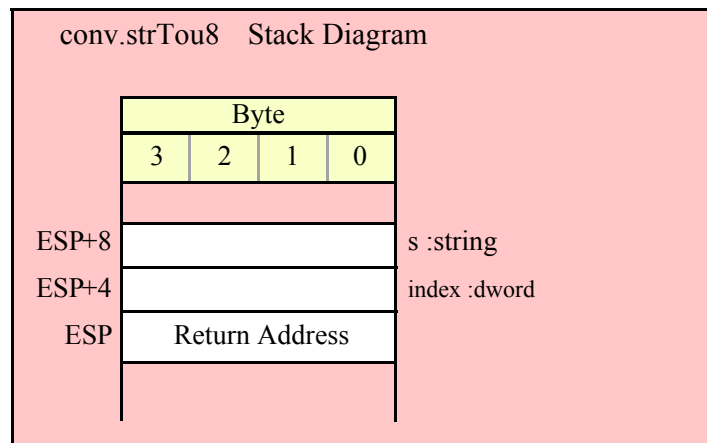
```
conv.strTou8( "abc12", 3 ); // "12" begins at offset 3
mov( al, hex12 );
```

HLA low-level calling sequence examples:

```
push( decValueStr );// Same as first example above
pushd( 0 );
call conv.strTou8;
mov( al, decNumericResult );
```

// Same as second example above

```
static
  str12 :string := "abc12";
.
.
.
push( str12 );// Note that str12 points at "abc12".
pushd( 3 );// Index to "12" in "abc12".
call conv.strTou8;
mov( al, dec12 );
```



```
procedure conv.strTou16( s:string; index:dword )
```

This function converts the sequence of characters starting at zero-based offset index within the string parameter s to a 16-bit unsigned integer. It returns the value (in the range 0..65535) in AX. Note that this function actually returns the zero-extended value in EAX, so you may use EAX if it is more convenient to do so.

HLA high-level calling sequence examples:

```
// The following will convert the characters at the beginning
// of "decValueStr" to numeric form:
```

```
conv.strTou16( hexValueStr, 0 );// Index=0 starts at beginning
mov( ax, wordVar );
```

```
// The following demonstrates using a non-zero index:
```

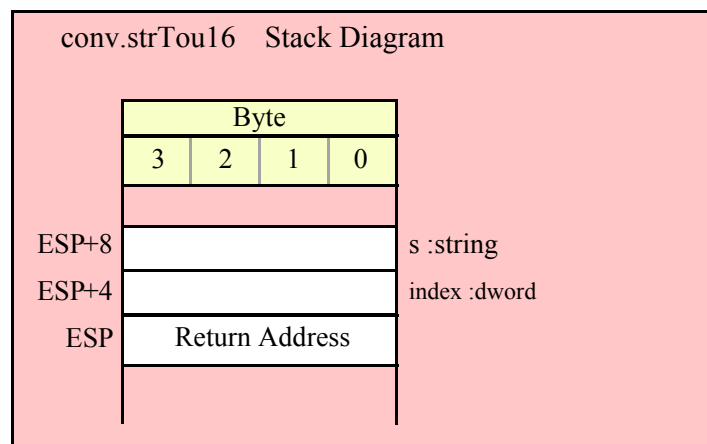
```
conv.strTou16( "abc1234", 3 ); // "1234" begins at offset 3
mov( ax, wordVar );
```

HLA low-level calling sequence examples:

```
push( decValueStr );// Same as first example above
pushd( 0 );
call conv.strTou16;
mov( ax, wordVar );
```

```
// Same as second example above
```

```
static
  str1200 :string := "abc1200";
.
.
.
push( str1200 );// Note that str1200 points at "abc1200".
pushd( 3 );// Index to "1200" in "abc1200".
call conv.strTou16;
mov( ax, wordVar );
```



**procedure conv.strTou32( s:string; index:dword )**

This function converts the sequence of characters starting at zero-based offset index within the string parameter s to a 32-bit unsigned integer. It returns the value (in the range 0..4294967295) in EAX.

HLA high-level calling sequence examples:

```
// The following will convert the characters at the beginning
// of "decValueStr" to numeric form:

conv.strTou32( decValueStr, 0 );// Index=0 starts at beginning
mov( eax, dwordVar );

// The following demonstrates using a non-zero index:

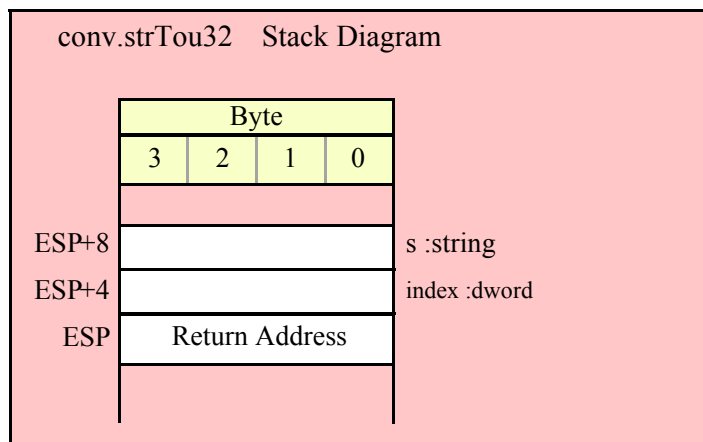
conv.strTou32( "abc12_345", 3 ); // "12_345" begins at offset 3
mov( eax, dwordVar );

HLA low-level calling sequence examples:

    push( decValueStr );// Same as first example above
    pushd( 0 );
    call conv.strTou32;
    mov( eax, dwordVar );

// Same as second example above

static
    str12345 :string := "abc012_345";
    .
    .
    .
push( str12345 );// Note that str12345 points at "abc-12_345".
pushd( 3 ); // Index to "012_345" in "abc012_345".
call conv.strTou32;
mov( eax, dwordVar );// dwordVar now contains 12,345.
```



```
procedure conv.strTou64( s:string; index:dword )
```

This function converts the sequence of characters starting at zero-based offset index within the string parameter s to a 64-bit unsigned integer. It returns the value (in the range 0.. 18446744073709551615) in EDX:EAX (EDX contains the H.O. dword).

HLA high-level calling sequence examples:

```
// The following will convert the characters at the beginning
// of "decValueStr" to numeric form:

conv.strTou64( decValueStr, 0 );// Index=0 starts at beginning
mov( eax, (type dword qwordVar[0]) );
mov( edx, (type dword qwordVar[4]) );

// The following demonstrates using a non-zero index:

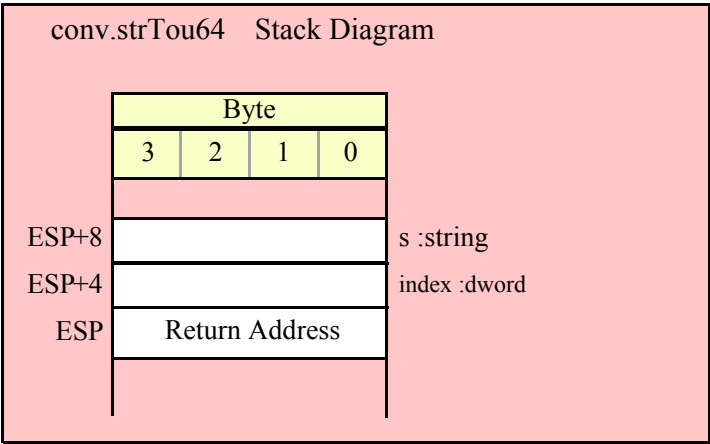
conv.strTou64( "a9123", 1 ); // "9123" begins at offset 1
mov( eax, (type dword qwordVar[0]) );
mov( edx, (type dword qwordVar[4]) );
```

HLA low-level calling sequence examples:

```
    push( decValueStr );// Same as first example above
    pushd( 0 );
    call conv.strTou64;
    mov( eax, (type dword qwordVar[0]) );
    mov( edx, (type dword qwordVar[4]) );

// Same as second example above

static
    strabc12 :string := "a9123";
    .
    .
    .
push( strabc12 );// Note that strabc12 points at "a9123".
pushd( 1 ); // Index to "-123" in "a9123".
call conv.strTou64;
mov( eax, (type dword qwordVar[0]) );
mov( edx, (type dword qwordVar[4]) );
```



```
procedure conv.strToul28( s:string; index:dword; var dest:lword )
```

This function converts the sequence of characters starting at zero-based offset index within the string parameter *s* to a 128-bit unsigned integer. It returns the value (in the range 0..340282366920938463463374607431768211456) in the parameter *l* that you pass by reference to this function.

HLA high-level calling sequence examples:

```
// The following will convert the characters at the beginning
// of "decValueStr" (index=0) to numeric form and store the
// 128-bit result into the lwordDest variable:
```

```
conv.strToul28( decValueStr, 0, lwordDest );
```

```
// The following demonstrates using a non-zero index:
```

```
conv.strToul28( "abc1234567890123456789", 3, lwordDest );
```

HLA low-level calling sequence examples:

```
// Option #1: lwordDest is a STATIC/READONLY/STORAGE
// variable:
```

```
push( decValueStr );// Same as first example above
pushd( 0 );
pushd( &lwordDest );
call conv.strToul28;
```

```
// Option #2: lwordDest is not a static object and
// a 32-bit register is available for use:
```

```
push( decValueStr );// Same as first example above
pushd( 0 );
lea( eax, lwordDest ); // Assuming EAX is available
push( eax );
call conv.strToul28;
```

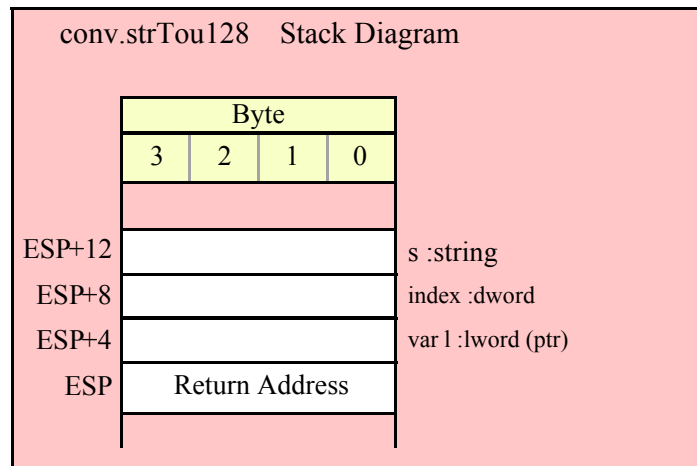
```
// Option #3: lwordDest is an automatic (var) object and
// no 32-bit registers are available for use:
```

```
push( decValueStr );// Same as first example above
pushd( 0 );
push( ebp );
add( @offset( lwordDest ), (type dword [esp]) );
call conv.strToul28;
```

```
// Option #4: lwordDest is a complex addressing mode object and
// no 32-bit registers are available for use:
```

```
push( decValueStr );// Same as first example above
pushd( 0 );
sub( 4, esp );
push( eax );
lea( eax, lwordDest );
mov( eax, [esp+4] );
pop( eax );
call conv.strToul28;
```





## 8.6 Floating Point Conversions

These functions convert between the three IEEE/Intel floating point formats and their string representation. There are two generic classes of these routines: those that convert their values to exponential/scientific notation and those that convert their string to a decimal format.

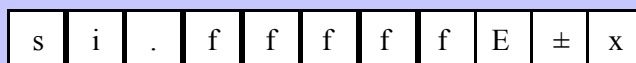
Note that the floating-point conversions do not insert underscores into the character sequences they produce. Therefore, these conversion functions ignore the internal underscores flag setting. If you wish to produce floating-point strings that have underscores between certain sets of digits, you should call one of these floating-point routines to do the basic conversion and then using other standard library routines to insert those underscores (or other character of your choosing) into the string.

**FPU Note:** The floating-point routines make use of the 80x86 x87 floating-point unit (FPU). Whenever you call one of these conversion routines, you must ensure that the CPU is operating in FPU mode (rather than MMX mode). If this is not the case, you should exit the MMX mode by executing an EMMS instruction prior to calling any of these conversion routines. As a general rule, you should use the SSE instructions rather than the MMX instructions and leave the CPU in FPU mode. Also note that the floating-point conversion routines make use of the FPU stack (probably as many as three elements, or so, just a guess) so you shouldn't leave any pending operations on the FPU stack when calling these conversion routines.

### 8.6.1 Exponential Floating-Point Conversions

The exponential floating-point conversion routines include conv.e32ToBuf, conv.e64ToBuf, conv.e80ToBuf, conv.e32ToStr, conv.e64ToStr, conv.e80ToStr, conv.a\_e32ToStr, conv.a\_e64ToStr, and conv.a\_e80ToStr. The \*Buf routines write the converted sequence of characters to memory, starting at the location pointed at by EDI. The \*Str routines store the converted character data into a string object.

The routines convert their values to a character sequence using scientific (exponential) notation. These routines each at least two parameters: the value to convert and the field width of the result that control how these functions format the output. These routines produce a string with the following format:



s is a space for positive values, '-' for negative values  
i represents the integer portion of the mantissa  
ffff represents the fractional portion of the mantissa  
x is one or more base-10 exponent digits.

The width parameter specifies the exact number of print positions the value will consume (i.e., the length of the resulting string). The first position holds the sign of the value. This is a space for positive values or a minus sign for negative values. If you do not want a leading space in front of positive values you can either store a "+" over the top of this space character (if the number is zero or positive) or you can call `str.trim` to remove any leading space.

The exponent field ('x') always uses the minimum number of digits to exactly represent the exponent. If the exponent is non-negative, then these routines preface the value with a '+'. If the exponent is negative, then these functions preface the exponent value with a '-' character.

The minimum field width you should specify is five. This allows one print position for the leading sign character, one digit for the mantissa, the "E", the exponent sign, and one exponent digit. Obviously, values greater than  $1E+9$  or less than  $1E-9$  will require additional print positions to handle the additional exponent digits.

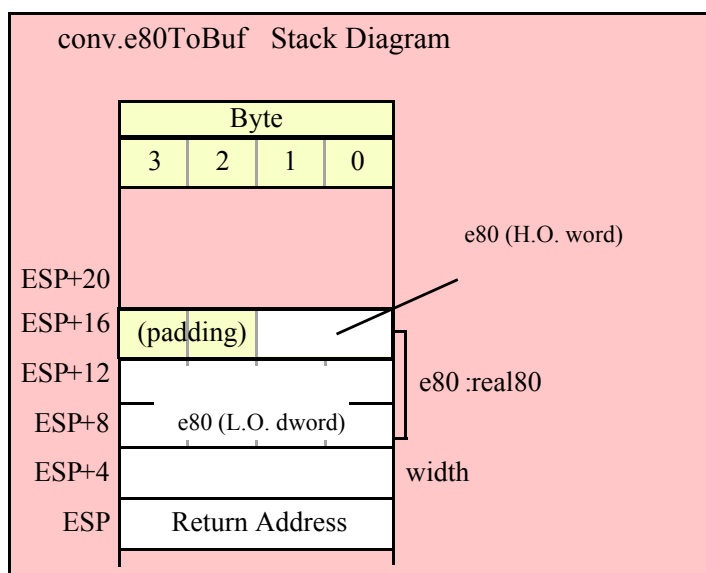
The number of fractional digits this routines produce is  $(width - 5) - \# \text{ exponent digits}$ . So you should choose your width according to the expected exponent size and the number of digits you would like to have to the right of the decimal point.

## 8.6.2 Floating Point Numeric to Buffer Conversions, Exponential Form

The floating point numeric to buffer conversion routines, `conv.e32ToBuf`, `conv.e64ToBuf`, and `conv.e80ToBuf`, translate the three different binary floating point formats to a sequence of characters that they store into sequential memory locations starting at the address held in the EDI register. They return EDI pointing at the first byte beyond the converted sequence. Note that these functions do not zero terminate the string; if you want a zero-terminated string, then store a zero at the byte pointed at by EDI upon return from these functions.

```
procedure conv.e80ToBuf
(
    e80:    real80;
    width:  uns32;
    var buffer: var in EDI
)
```

This function converts the 80-bit extended precision e80 value to its character representation using exponential/scientific notation. This function stores the resulting conversion starting at the address specified in EDI. It is the caller's responsibility to ensure that sufficient memory is available at this starting address. On return, EDI will point at the first byte beyond the converted string. Note that the 80-bit extended precision format supports approximately 18 decimal digits of precision. Therefore, any digits beyond the 18<sup>th</sup> significant digit will contain garbage. Hence, your choice of width should not produce more than 18 mantissa digits.

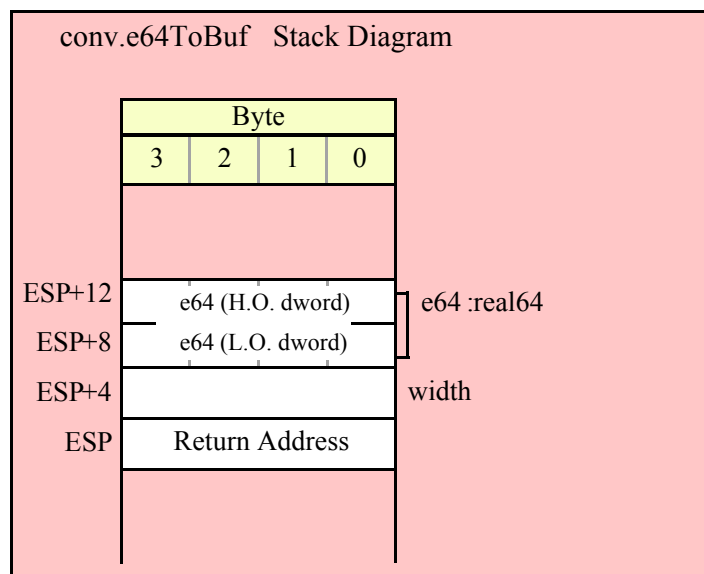


```

procedure conv.e64ToBuf
(
    e64:    real64;
    width: uns32;
    var buffer: var in EDI
)

```

This function converts the 64-bit extended precision e64 value to its character representation using exponential/scientific notation. This function stores the resulting conversion starting at the address specified in EDI. It is the caller's responsibility to ensure that sufficient memory is available at this starting address. On return, EDI will point at the first byte beyond the converted string. Note that the 64-bit extended precision format supports approximately 15 decimal digits of precision. Therefore, any digits beyond the 15<sup>th</sup> significant digit will contain garbage. Hence, your choice of width should not produce more than 15 mantissa digits.

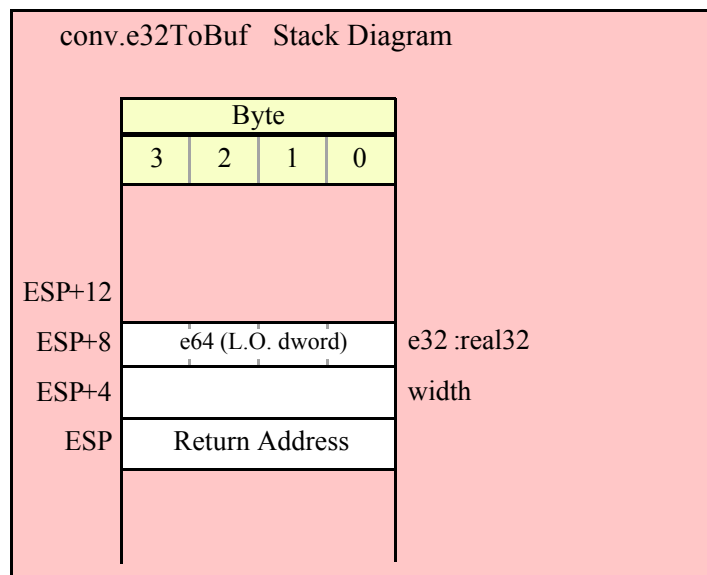


```

procedure conv.e32ToBuf
(
    e32:    real32;
    width: uns32;
    var buffer: var in EDI
)

```

This function converts the 32-bit extended precision e32 value to its character representation using exponential/scientific notation. This function stores the resulting conversion starting at the address specified in EDI. It is the caller's responsibility to ensure that sufficient memory is available at this starting address. On return, EDI will point at the first byte beyond the converted string. Note that the 32-bit single precision format supports approximately 6-7 decimal digits of precision. Therefore, any digits beyond the seventh significant digit will contain garbage. Hence, your choice of width should not produce more than seven mantissa digits.



### 8.6.3 Floating Point Numeric to String Conversions, Exponential Form

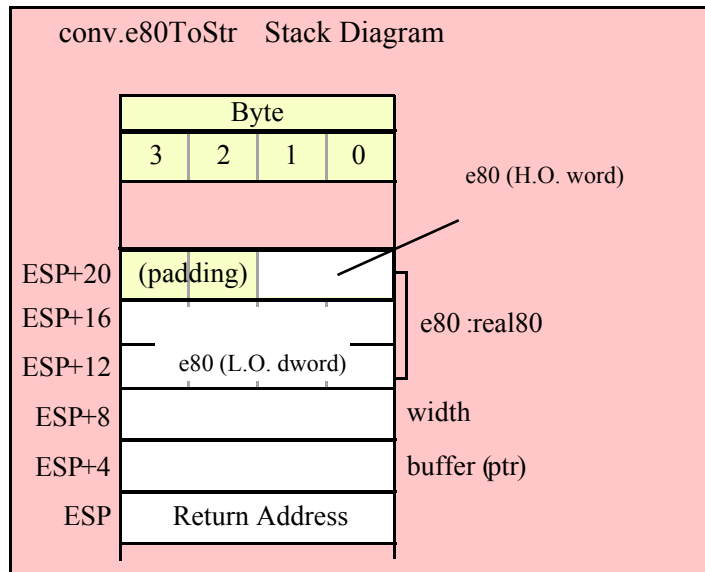
The floating point numeric to string conversion routines translate the three different binary floating point formats to their string representation. The standard ("unadorned") functions store the string data into a string object that you pass as a parameter to the function. That string object must be preallocated and large enough to receive the string result (else a string overflow occurs). The "adorned" functions, whose names begin with "a\_" automatically allocate storage on the heap, store the converted string into that heap object, and then return a pointer to the newly allocated string in the EAX register (it is the caller's responsibility to free the storage when it is no longer needed).

```

procedure conv.e80ToStr
(
    e80:      real80;
    width:   uns32;
    buffer:  string
)

```

This function converts the 80-bit extended precision e80 value to its string representation using exponential/scientific notation. This function stores the resulting string in the buffer variable whose `MaxLength` field must be at least `width` or this function will raise an exception. Note that the 80-bit extended precision format supports approximately 18 decimal digits of precision. Therefore, any digits beyond the 18<sup>th</sup> significant digit will contain garbage. Hence, your choice of `width` should not produce more than 18 mantissa digits.

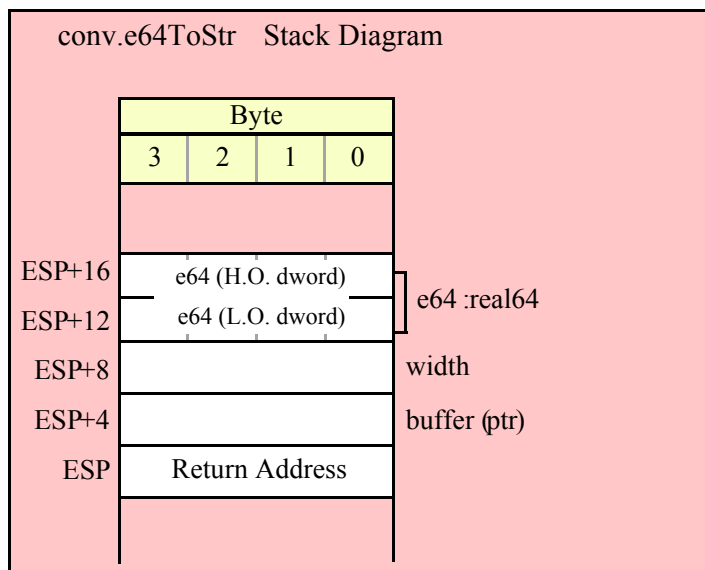


```

procedure conv.e64ToStr
(
    e64:      real64;
    width:   uns32;
    buffer:  string
)

```

This function converts the 64-bit double precision e64 value to its string representation using exponential/scientific notation. This function stores the resulting string in the buffer variable whose `MaxLength` field must be at least `width` or this function will raise an exception. Note that the 64-bit double precision format supports approximately 15 decimal digits of precision. Therefore, any digits beyond the 15<sup>th</sup> significant digit will contain garbage. Hence, your choice of `width` should not produce more than 15 mantissa digits.

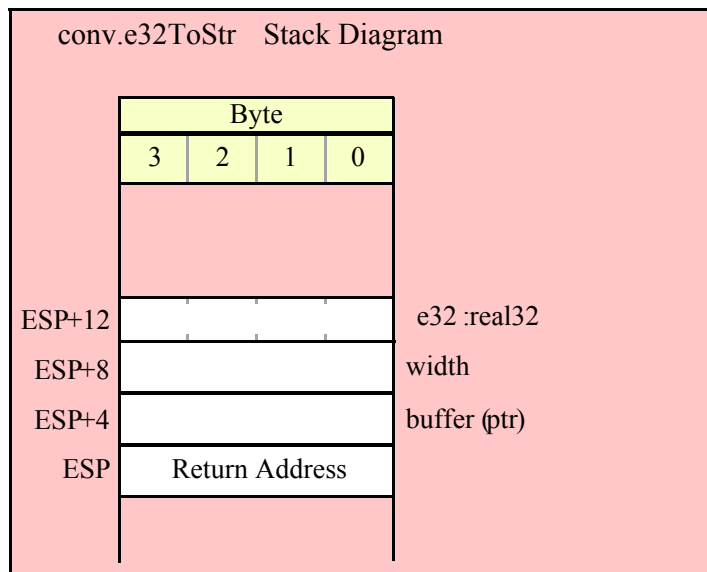


```

procedure conv.e32ToStr
(
    e32:  real32;
    width:uns32;
    buffer:string
)

```

This function converts the 32-bit single precision e32 value to its string representation using exponential/scientific notation. This function stores the resulting string in the buffer variable whose MaxLength field must be at least width or this function will raise an exception. Note that the 32-bit single precision format supports approximately 6-7 decimal digits of precision. Therefore, any digits beyond the seventh significant digit will contain garbage. Hence, your choice of width should not produce more than seven mantissa digits.

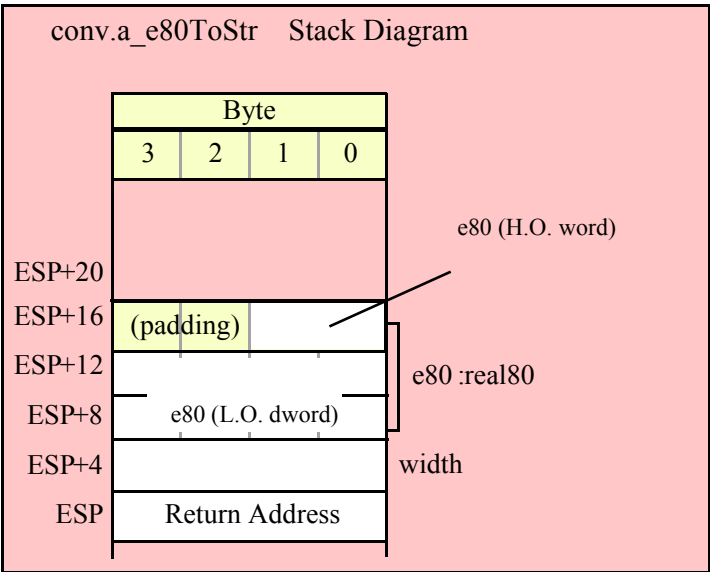


```

procedure conv.a_e80ToStr
(
    e80:      real80;
    width:    uns32
); @returns( "eax" );

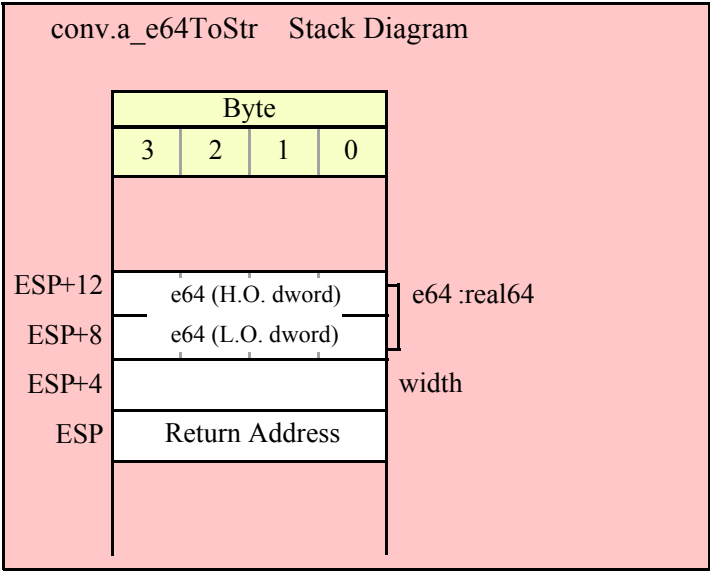
```

This function converts the 80-bit extended precision e80 value to its string representation using exponential/scientific notation. This function stores the resulting string in storage it allocates on the heap and returns a pointer to that string in the EAX register. It is the caller's responsibility to deallocate this storage when they are done with it. Note that the 80-bit extended precision format supports approximately 18 decimal digits of precision. Therefore, any digits beyond the 18<sup>th</sup> significant digit will contain garbage. Hence, your choice of width should not produce more than 18 mantissa digits.



```
procedure conv.a_e64ToStr
(
    e64:      real64;
    width:    uns32
); @returns( "eax" );
```

This function converts the 64-bit double precision e64 value to its string representation using exponential/scientific notation. This function stores the resulting string in storage it allocates on the heap and returns a pointer to that string in the EAX register. It is the caller's responsibility to deallocate this storage when they are done with it. Note that the 64-bit double precision format supports approximately 15 decimal digits of precision. Therefore, any digits beyond the 15<sup>th</sup> significant digit will contain garbage. Hence, your choice of width should not produce more than 15 mantissa digits.

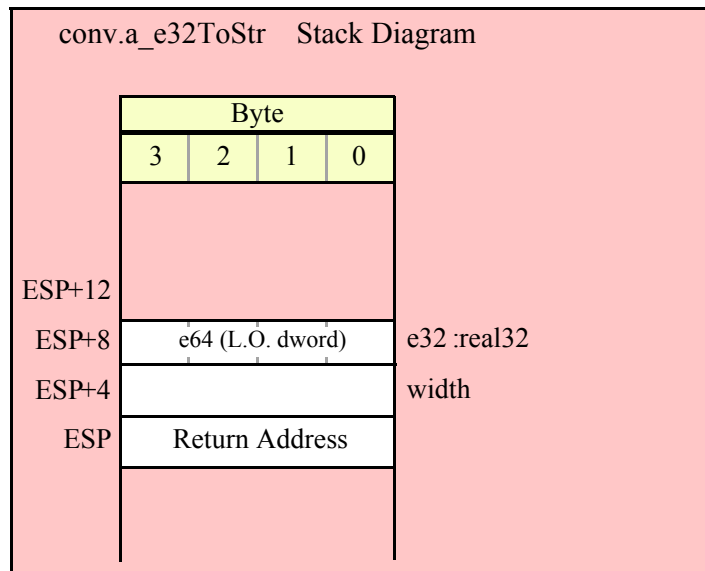


```

procedure conv.a_e32ToStr
(
    e32:          real32;
    width:        uns32
); @returns( "eax" );

```

This function converts the 32-bit single precision e32 value to its string representation using exponential/scientific notation. This function stores the resulting string in storage it allocates on the heap and returns a pointer to that string in the EAX register. It is the caller's responsibility to deallocate this storage when they are done with it. Note that the 32-bit single precision format supports approximately 6-7 decimal digits of precision. Therefore, any digits beyond the seventh significant digit will contain garbage. Hence, your choice of width should not produce more than seven mantissa digits.



### 8.6.4 Floating Point Numeric to Character Conversions, Decimal Form

Although scientific (exponential) notation is the most general display format for real numbers, real numbers you display in this format are difficult to read. Therefore, the standard library conversions module also provides a set of functions that convert real values to their decimal string equivalent. Although you cannot (practically) use these decimal conversion routines for all real values, they are applicable to a wide variety of common numbers you will use in your programs.

These functions all have at least four parameters: the real value to convert, the width of the converted value, the number of digit positions to the right of the decimal point, and a padding character. These functions convert their values to the following string format:



s	i	i	i	.	f	f	f	f	f	f
---	---	---	---	---	---	---	---	---	---	---

s is a space for positive values, '-' for negative values  
i represents the integer portion of the mantissa  
ffff represents the fractional portion of the mantissa

The width parameter specifies the length of the resulting string. This value must be less than or equal to the destination string's MaxLength value or these functions will raise an exception. The decimalpts parameter to these functions specify the number of digits to the right of the decimal point. If this parameter contains zero, then these functions display the value as an integer (no fractional digits and no decimal point). If this parameter is non-zero, then these routines produce the specified number of decimal digits along with a decimal point.

The width parameter specifies the total size of the resulting string. If decimalpts is zero, then the width value must be at least one greater than the number of digits that appear to the left of the decimal point (the extra position is for the sign character. If the decimalpts parameter is non-zero, then width must be at least (decimalpts + 2 + # integer digits). If width is not sufficiently large, then these functions produce a string containing width "#" characters to denote a conversion error.

If the width value is sufficiently large and the decimalpts sufficiently small then these routines will fill the extra print positions using the fill character you pass as a parameter. For example, if you convert the value -1.5 with a width of six, a decimalpts value of two, and a fill character of "\*" these routines produce the string "\*-1.50".

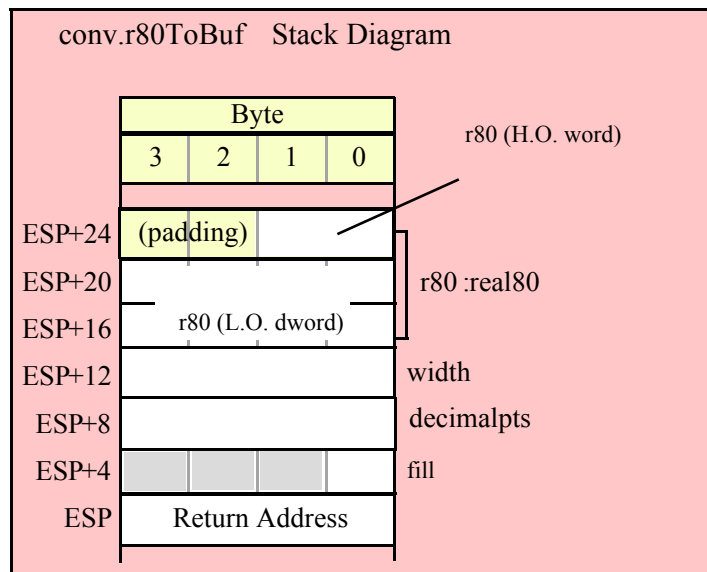
The floating point numeric to buffer conversion routines, conv.r32ToBuf, conv.r64ToBuf, and conv.r80ToBuf, translate the three different binary floating point formats to a sequence of characters that they store into sequential memory locations starting at the address held in the EDI register. They return EDI pointing at the first byte beyond the converted sequence. Note that these functions do not zero terminate the string; if you want a zero-terminated string, then store a zero at the byte pointed at by EDI upon return from these functions.

```

procedure conv.r80ToBuf
(
    r80:      real80;
    width:    uns32;
    decimalpts: uns32;
    fill:     char;
var    buffer: var in edi
)

```

This function converts the 80-bit extended precision r80 value to its string representation using decimal notation. This function stores the resulting character sequence into sequential memory locations starting at the address held in the EDI register. They return EDI pointing at the first byte beyond the converted sequence. Note that these functions do not zero terminate the string; if you want a zero-terminated string, then store a zero at the byte pointed at by EDI upon return from these functions. Note that the 80-bit single precision format supports approximately 18 decimal digits of precision. Therefore, any digits beyond the 18<sup>th</sup> significant digit will contain garbage. Hence, your choice of width should not produce more than 18 mantissa digits. Do keep in mind that the average person has trouble comprehending value with more than six or seven digits. For values that are routinely outside this range you may want to use exponential form to display the number with a limited number of significant digits.

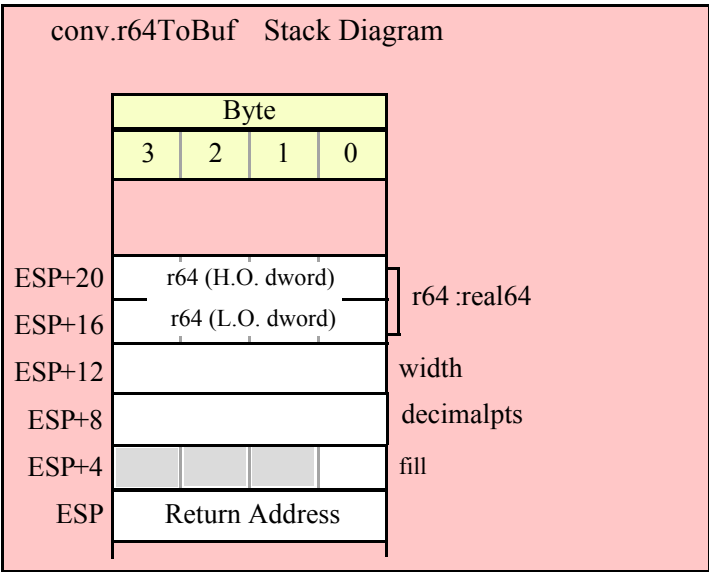


```

procedure conv.r64ToBuf
(
    r64:          real64;
    width:       uns32;
    decimalpts: uns32;
    fill:        char;
    varbuffer:   var in edi
)

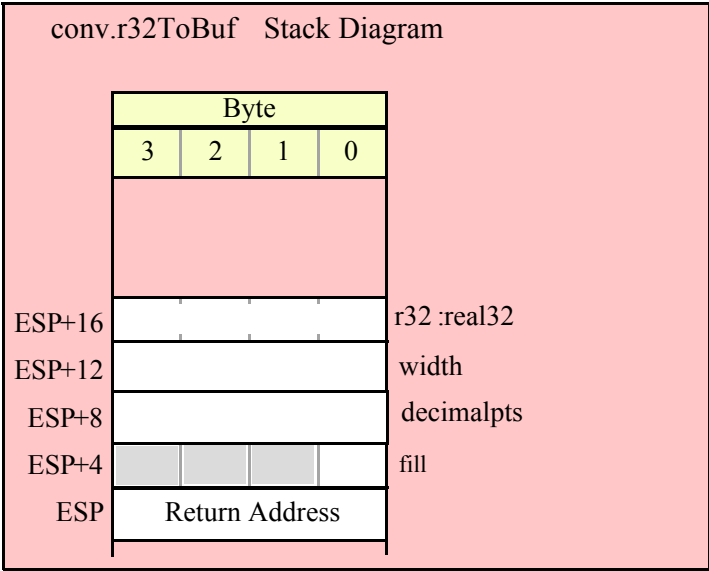
```

This function converts the 64-bit double precision r64 value to its string representation using decimal notation. This function stores the resulting character sequence into sequential memory locations starting at the address held in the EDI register. They return EDI pointing at the first byte beyond the converted sequence. Note that these functions do not zero terminate the string; if you want a zero-terminated string, then store a zero at the byte pointed at by EDI upon return from these functions. Note that the 64-bit single precision format supports approximately 15 decimal digits of precision. Therefore, any digits beyond the 15<sup>th</sup> significant digit will contain garbage. Hence, your choice of width should not produce more than 15 mantissa digits. Do keep in mind that the average person has trouble comprehending value with more than six or seven digits. For values that are routinely outside this range you may want to use exponential form to display the number with a limited number of significant digits.



```
procedure conv.r32ToBuf
(
    r32:      real32;
    width:    uns32;
    decimalpts: uns32;
    fill:     char;
    varbuffer: var in edi
)
```

This function converts the 32-bit single precision r32 value to its string representation using decimal notation. This function stores the resulting Note that the 32-bit single precision format supports approximately 6-7 decimal digits of precision. Therefore, any digits beyond the seventh significant digit will contain garbage. Hence, your choice of width should not produce more than seven mantissa digits.



## 8.6.5 Floating-Point Numeric to String Conversions, Decimal Form

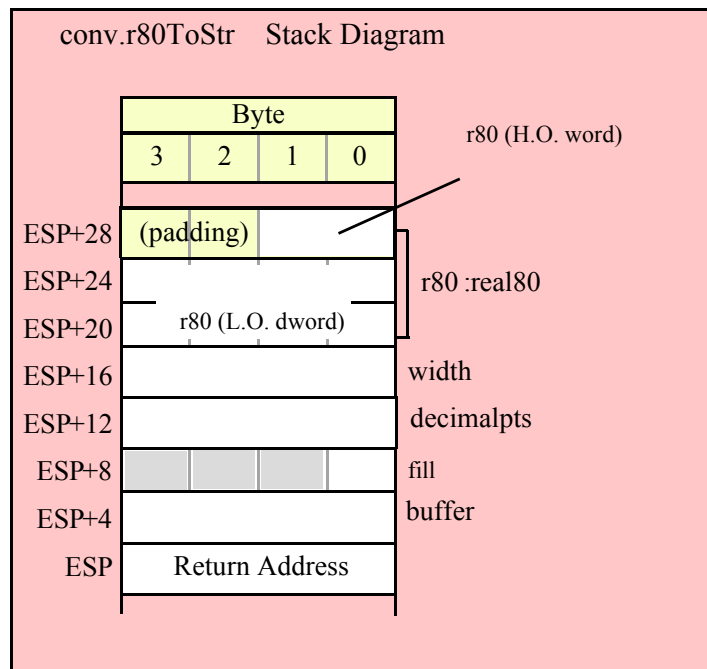
The floating point numeric to string conversion routines translate the three different binary floating point formats to their string representation. The standard ("unadorned") functions store the string data into a string object that you pass as a parameter to the function. That string object must be preallocated and large enough to receive the string result (else a string overflow occurs). The "adorned" functions, whose names begin with "a\_" automatically allocate storage on the heap, store the converted string into that heap object, and then return a pointer to the newly allocated string in the EAX register (it is the caller's responsibility to free the storage when it is no longer needed).

```

procedure conv.r80ToStr
(
    r80:      real80;
    width:    uns32;
    decimalpts: uns32;
    fill:     char;
    buffer:   string
)

```

This function converts the 80-bit extended precision r80 value to its string representation using decimal notation. This function stores the resulting string in the buffer variable whose MaxLength field must be at least width or this function will raise an exception. Note that the 80-bit single precision format supports approximately 18 decimal digits of precision. Therefore, any digits beyond the 18<sup>th</sup> significant digit will contain garbage. Hence, your choice of width should not produce more than 18 mantissa digits. Do keep in mind that the average person has trouble comprehending value with more than six or seven digits. For values that are routinely outside this range you may want to use exponential form to display the number with a limited number of significant digits.



```

procedure conv.r64ToStr
(

```

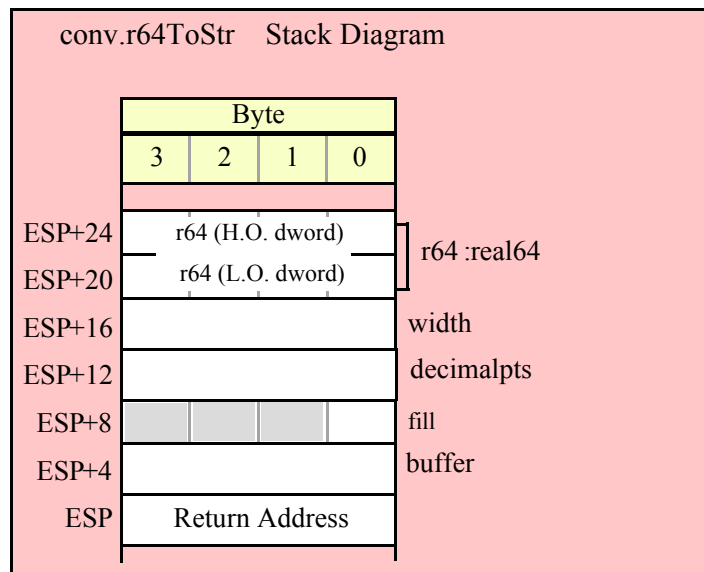
```

    r64:      real64;
    width:    uns32;

```

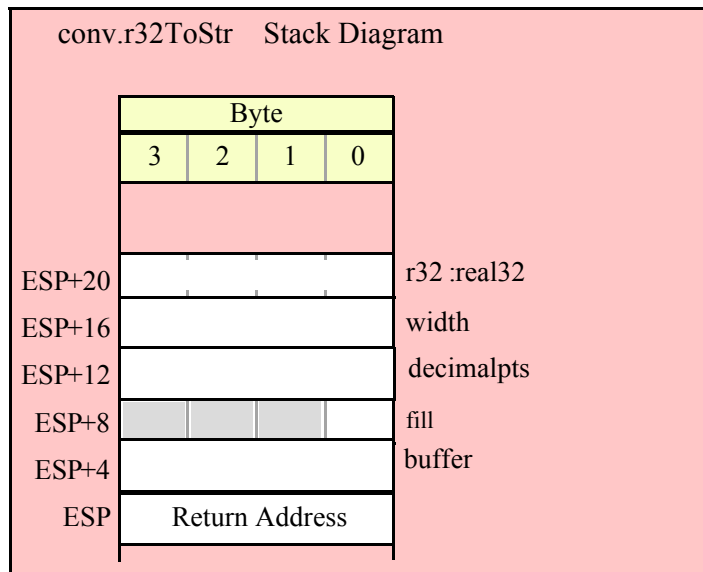
```
decimalpts: uns32;
fill:      char;
buffer:    string
)
```

This function converts the 64-bit double precision r64 value to its string representation using decimal notation. This function stores the resulting string in the buffer variable whose MaxLength field must be at least width or this function will raise an exception. Note that the 64-bit single precision format supports approximately 15 decimal digits of precision. Therefore, any digits beyond the 15<sup>th</sup> significant digit will contain garbage. Hence, your choice of width should not produce more than 15 mantissa digits. Do keep in mind that the average person has trouble comprehending value with more than six or seven digits. For values that are routinely outside this range you may want to use exponential form to display the number with a limited number of significant digits.



```
procedure conv.r32ToStr
(
    r32:          real32;
    width:        uns32;
    decimalpts:   uns32;
    fill:         char;
    buffer:       string
)
```

This function converts the 32-bit single precision r32 value to its string representation using decimal notation. This function stores the resulting string in the buffer variable whose MaxLength field must be at least width or this function will raise an exception. Note that the 32-bit single precision format supports approximately 6-7 decimal digits of precision. Therefore, any digits beyond the seventh significant digit will contain garbage. Hence, your choice of width should not produce more than seven mantissa digits.

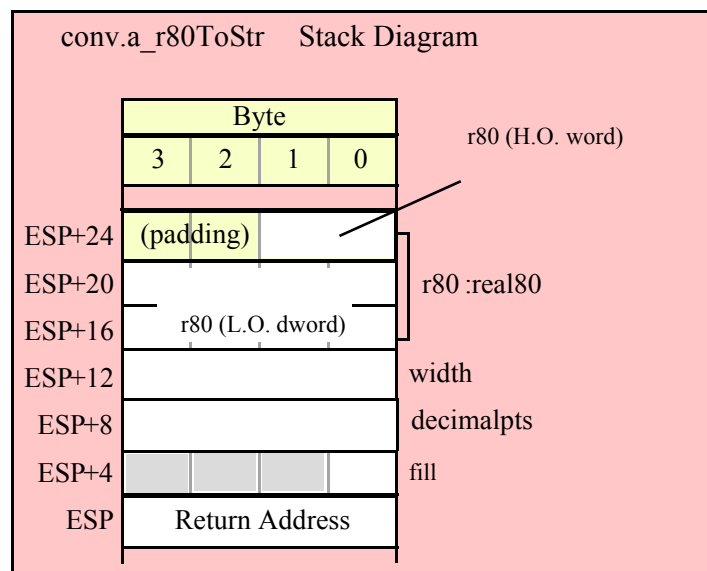


```

procedure conv.a_r80ToStr
(
    r80:      real80;
    width:    uns32;
    decimalpts: uns32;
    fill:     char
); @returns( "eax" );

```

This function converts the 80-bit extended precision r80 value to its string representation using decimal notation. This function stores the resulting string in storage it allocates on the heap and returns a pointer to that string in the EAX register. It is the caller's responsibility to deallocate this storage when they are done with it.. Note that the 80-bit single precision format supports approximately 18 decimal digits of precision. Therefore, any digits beyond the 18<sup>th</sup> significant digit will contain garbage. Hence, your choice of width should not produce more than 18 mantissa digits. Do keep in mind that the average person has trouble comprehending value with more than six or seven digits. For values that are routinely outside this range you may want to use exponential form to display the number with a limited number of significant digits.

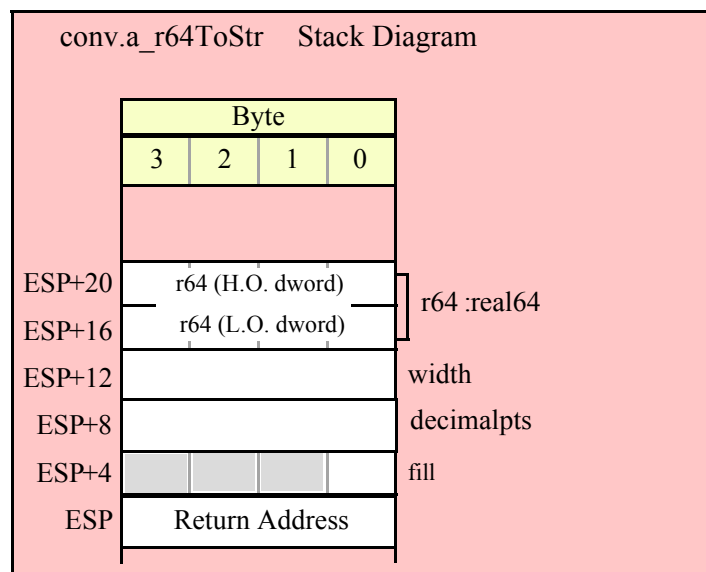


```

procedure conv.a_r64ToStr
(
    r64:      real64;
    width:    uns32;
    decimalpts: uns32;
    fill:     char
); @returns( "eax" );

```

This function converts the 64-bit double precision r64 value to its string representation using decimal notation. This function stores the resulting string in storage it allocates on the heap and returns a pointer to that string in the EAX register. It is the caller's responsibility to deallocate this storage when they are done with it. Note that the 64-bit single precision format supports approximately 15 decimal digits of precision. Therefore, any digits beyond the 15<sup>th</sup> significant digit will contain garbage. Hence, your choice of width should not produce more than 15 mantissa digits. Do keep in mind that the average person has trouble comprehending value with more than six or seven digits. For values that are routinely outside this range you may want to use exponential form to display the number with a limited number of significant digits.

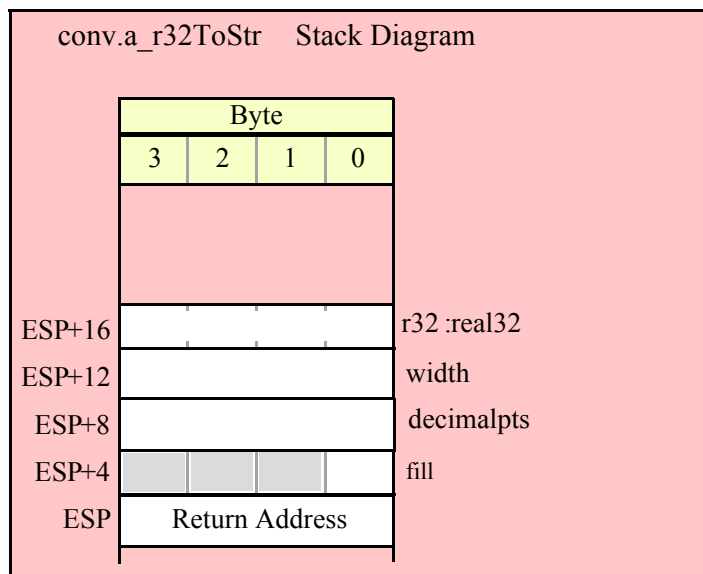


```

procedure conv.a_r32ToStr
(
    r32:      real32;
    width:    uns32;
    decimalpts: uns32;
    fill:     char
); @returns( "eax" );

```

This function converts the 32-bit single precision r32 value to its string representation using decimal notation. This function stores the resulting string in storage it allocates on the heap and returns a pointer to that string in the EAX register. It is the caller's responsibility to deallocate this storage when they are done with it. Note that the 32-bit single precision format supports approximately 6-7 decimal digits of precision. Therefore, any digits beyond the seventh significant digit will contain garbage. Hence, your choice of width should not produce more than seven mantissa digits.



### 8.6.6 Floating Point String/Buffer to Numeric Conversions

The floating-point string to numeric routines convert characters found in a character sequence to an 80-bit IEEE floating-point format. There are two versions – `conv.atof` and `conv.strToFlt`. `conv.atof` operates on an arbitrary sequence of characters in memory and `conv.strToFlt` operates on a string variable.

These routines will skip over any leading underscore and delimiter characters (specified by the internal delimiters character set, see the discussion of `conv.setDelimiters` and `conv.getDelimiters` for details). These functions will convert all characters in the sequence until encountering an illegal floating-point character (decimal digits, a decimal point, 'e' or 'E', and an optional sign for the exponent and mantissa). If the first non-acceptable character is not the end of string or a delimiter character, these functions will raise a conversion exception. If the character is not a valid 7-bit ASCII character, these functions will raise an illegal character exception.

The `conv.strToFlt` functions has two parameters: a string object and an index into that string. Numeric conversion because at the zero-based character position specified by the index parameter. For example, the invocation

```
conv.strToFlt( someStr, 5 );
```

begins the conversion starting with the sixth character (index 5) in `someStr`. These functions will raise an "index out of range" exception if the supplied index is greater than the size of the string the first parameter specifies. They will return a null pointer reference exception if the string parameter is NULL (they will return an illegal memory access exception if the first parameter is not a valid pointer).

These functions always convert their strings to an 80-bit floating-point value and leave that value sitting on the top of the FPU stack (ST0). If you want the conversion to a 32-bit or 64-bit floating-point format, then use the `fstp` instruction to store the result in whatever destination format you desire (`real32`, `real64`, or `real80`).

```
procedure conv.atof( bufptr: dword in esi ); @returns( "st0" );
```

This routine assumes that ESI is pointing at a sequence of characters that represents a floating point number. The characters are converted to numeric form and the result is returned in ST0. ESI is left pointing at the first character beyond the converted characters. This function raises an exception if the value begins with something other than a standard numeric, '-', or delimiter character or ends with something other than a standard delimiter character (or the end of string).

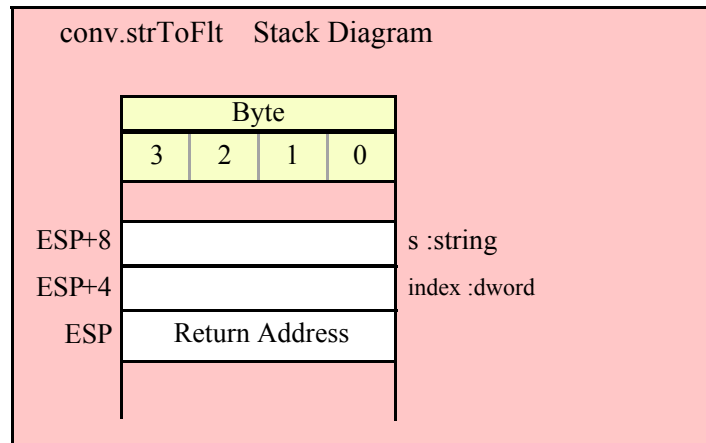
This routine accepts floating point input in either decimal or exponential form.



```
procedure conv.strToFlt( s:string; index:dword ); @returns( "st0" );
```

This function converts the sequence of characters starting at position `index` in `s` to the equivalent extended precision floating point value and it leaves the result in `ST0`. This function raises an exception if the value begins with something other than a standard numeric, '-', or delimiter character, or ends with something other than a standard delimiter character (or the end of string).

This routine accepts floating point input in either decimal or exponential form.



## 8.6.7 Roman Numeral Conversion

```
procedure conv.roman( Arabic:uns32; rmn:string )
```

This procedure converts the specified integer value (`Arabic`) into a string that contains the Roman numeral representation of the value. Note that this routine only converts integer values in the range 1..3,999 to Roman numeral form. Since ASCII text doesn't allow overbars (that multiply roman digits by 1,000), this function doesn't handle really large Roman numbers. A different character set would be necessary for that.

```
conv.a_roman( Arabic:uns32 )
```

Just like the routine above, but this one allocates storage for the string and returns a pointer to the string in the `EAX` register.



## 9 Coroutines Module (coroutines.hhf)

HLA provides a powerful coroutines class that lets you easily use coroutines in your programs. The coroutine class provides three procedures and methods you can use to initialize a coroutine, transfer control between coroutines, and free up the storage associated with a coroutine when it completes execution. The coroutine class also has several data fields, but you should treat these as private fields and never disturb their values.

In addition to these class procedures and methods, the coroutine package provides a `coret` procedure that is useful for returning from a coroutine to whomever "cocalled" the coroutine. This makes it very easy to implement *Generators* using coroutines.

Finally, the coroutine module provides a special coroutine variable, `mainPgm`, that you can use to cocall the "coroutine" corresponding to the main HLA program.

### 9.1 The Coroutine Module

To use the coroutine functions in your application, you will need to include one of the following statements at the beginning of your HLA application:

```
#include( "coroutines.hhf" )
or
#include( "stdlib.hhf" )
```

### 9.2 The Coroutine Class Definition

Here's the definition of the coroutine class data type:

```
// Note: the original declaration was "coroutine"
// but this has been deprecated. The following text
// equate is for legacy code. Someday, this declaration
// will go away.

const
    coroutine:text := "coroutine_t";

type
    coroutine_t:
        class

            var
                CurrentSP:      dword;
                Stack:          dword;
                ExceptionContext: dword;
                LastCaller:      dword;

            procedure cocall;
                @external( "COR_COCALL" );

            procedure create( size:uns32; theProc:procedure );
                @external( "COR_CREATE" );

            method cofree;
                @external( "COR_COFREE" );

        endclass;
```

The data fields are all private fields to this class, your applications should not modify these fields. In addition to the two procedures and the method in this class, the `coroutines.hhf` header file also defines a single external procedure and an external coroutine variable:

```
procedure coret; @external( "COR_CORET" );

static
    mainPgm_coroutine:coroutine_t; @external( "MainPgmCoroutine__hla_" );
```

## 9.3 Coroutine Functions

```
procedure coroutine_t.create( size:uns32; theProc:procedure );
```

`coroutine_t.create` is the typical HLA class constructor for the coroutine class. Since this is a class procedure, you can call create one of two different ways:

(1) You can call it via the statement `"coroutine_t.create( size, proc);"` This form assumes that you wish to create a dynamic coroutine object on the heap. When called this way, the `coroutine_t.create` procedure allocates storage for a coroutine object on the heap and returns a pointer to this new coroutine object in the ESI register. Otherwise it behaves identically to the second form of the `coroutine_t.create` procedure.

(2) You can call `coroutine_t.create` using an invocation of the form `"objectName.create( size, proc);"` where `"objectName"` is the name of a `coroutine_t` variable or a pointer to a `coroutine_t` object (that, presumably, has been initialized with a valid pointer to a `coroutine_t` object). Do be aware that this form of the call loads ESI with the address of the `coroutine_t` object. On return, ESI will contain this new value.

Either form of the call to create will initialize the `coroutine_t` object, allowing subsequent cocalls to the `coroutine_t` object.

Coroutines execute using their own stack (independent of other coroutine stacks and independent of the stack the main program uses). The `size` parameter specifies the number of bytes of stack space to reserve for the coroutine. A good minimum value for a coroutine stack is between 256 and 1,024 bytes. If the coroutine allocates lots of local/automatic variables, or calls other procedures that allocate lots of local/automatic storage, you will need to allocate a larger stack as appropriate. Likewise, if your coroutine calls procedures that are recursive, additional stack space may be necessary.

The `theProc` parameter is a pointer to a procedure. This procedure is the code that will execute when you cocall this coroutine. The only thing special about the procedure is that it should never be possible to return to the procedure's caller by executing a RET instruction. You exit coroutine using the `coroutine_t.cocall` procedure or the `coroutine_t.coret` procedure. If your code accidentally "falls off the end of the procedure" or otherwise attempts to return to the caller via a RET instruction, the coroutine will go into a special state in which any attempt to cocall it forces an immediate return by the coroutine to the caller.

Object declarations for examples:

```
static
    ptrToCoroutine:pointer to coroutine;
    staticCoroutine:coroutine_t;
```

HLA high-level calling sequence examples:

```
coroutine_t.create( 1024, &myCoroutineProc );
mov( esi, ptrToCoroutine );
staticCoroutine.create( 1024, &anotherProc );
```

HLA low-level calling sequence examples:

```
pushd( 1024 );
pushd( &myCoroutineProc );
mov( NULL, esi );// Tells create to allocate storage
call coroutine_t.create;
```

```

mov( esi, ptrToCoroutine );

pushd( 1024 );
pushd( &anotherProc );
lea( esi, staticCoroutine );
call coroutine.create;

```

#### **procedure coroutine\_t.cocall();**

*coroutine\_t.cocall* is the mechanism you use to invoke a coroutine. Note that this is a procedure for performance reasons. You should never invoke the static procedure *coroutine\_t.cocall* as this will raise a run-time exception. Instead, you should always invoke this procedure using an object invocation of the form "objectName.cocall();" This will switch the thread of execution from the current coroutine (or the main program) to the coroutine code associated with "objectName". Note that coroutines rarely begin execution at the first statement of the procedure associated with the coroutine (in fact, this happens exactly once, when you invoke the coroutine for the very first time).

The cocall mechanism provides the standard way of leaving a coroutine. Cocalling some other coroutine switches the execution context from the current coroutine to that other coroutine. The next time some code cocalls a coroutine that leaves via cocall, execution continues with the first statement following the cocall (it's almost as though you had called that other coroutine using a CALL instruction).

HLA high-level calling sequence examples:

```

ptrToCoroutine.cocall();
staticCoroutine.cocall();

```

HLA low-level calling sequence examples:

```

mov( ptrToCoroutine, esi );
call coroutine_t.cocall;

lea( esi, staticCoroutine );
call coroutine_t.cocall;

```

#### **method coroutine\_t.cofree();**

When you are done with a coroutine, you should call the *coroutine\_t.cofree* method to free up the stack space associated with that coroutine. You must not call *coroutine\_t.cofree* from inside the coroutine you're cleaning up since it still needs its stack to transfer control to some other coroutine.

HLA high-level calling sequence examples:

```

ptrToCoroutine.cofree();
staticCoroutine.cofree();

```

HLA low-level calling sequence examples:

```

mov( ptrToCoroutine, esi );
mov( [esi], edi );
call( [edi+@offset(coroutine_t.cofree)] );

lea( esi, staticCoroutine );
mov( [esi], edi );
call( [edi+@offset(coroutine_t.cofree)] );

```

**method coret();**

*coret* is nearly identical to *coroutine\_t.cocall* with two major exceptions. First, note that this procedure is not a member of the *coroutine\_t* class. Therefore, you do not specify an object name in front of the call to the *coret* procedure. Second, *coret* returns control to whomever cocalled the current coroutine. The current coroutine does not have to know who called it; *coret* figures this out and cocalls the appropriate coroutine.

Note that *coret* is not a "return" in the usual sense that the coroutine completes execution upon calling *coret*. *coret* is identical to a *coroutine\_t.cocall* to the coroutine that called the current coroutine. In particular, after a coroutine returns to another, any future cocalls to this coroutine will continue execution with the first statement following the *coret* call.

HLA high-level calling sequence examples:

```
coret();
```

HLA low-level calling sequence examples:

```
call coret;
```

**static mainPgm:coroutine\_t;**

This is a special *coroutine\_t* variable that contains the control information for the main program. If, inside a coroutine, you wish to cocall the main program, just use a cocall of the form "MainPgm.cocall();" and control in the main program will continue at the point of the last cocall executed in the main program. (Note: the term "main program" here does not imply that the cocall has to be in the actual main program of an HLA program, it simply refers to the thread of execution that starts in the main program. Your main program can call a procedure that transfers control to some coroutine via cocall. *MainPgm.cocall* will transfer control back into that procedure.)

## 10 Character Sets (cset.hhf)

The HLA Standard Library contains several routines that provide the power of the HLA compile-time character set facilities at run-time (i.e., within your programs).

HLA uses a 128-bit bitmap (16 consecutive bytes) to implement sets of seven-bit ASCII characters. This has a very important implication: you cannot pass byte values greater than \$7F to a character set function. Currently, the HLA Standard Library routines do not check for values out of range (for performance reasons). In the future, this checking may be added as a compilable option. For the time being, however, it is your responsibility to verify that all character values are in the range #0..#\$7F (and, in general, #0 is an exceeding bad value to specify in many cases since the null character terminates strings).

The bitmap consists of 128 consecutive bits numbered 0..127. If a bit in a character set is one, then the corresponding character (whose ASCII code matches the bit number) is a member of the character set. Conversely, if a bit is zero, the corresponding character is not a member of the set.

Note that many routines pass character sets by value. This means you can pass HLA character set constants as parameters to these procedures/functions. HLA emits four MOV (doubleword) instructions to copy a character set by value, so passing character sets by value is not horribly inefficient (though not quite as fast as a 32-bit integer!).

**Warning:** All of the character set routines are members of the `cs` namespace. This means you cannot use the name `cs` within your programs. (`cs` is a common character set name that lazy programmers use; sorry, it's already been taken!)

The following sections describe each of the character set routines in the HLA Standard Library.

**A Note About Thread Safety:** The routines in this module are all thread safe.

**Note about stack diagrams:** this documentation includes stack diagrams for those functions that pass parameters on the stack. To conserve space, this documentation does not include a stack diagram for any function that does not pass data on the stack (that is, only a return address appears on the stack).

### 10.1 Predicates (tests)

Although the "returns" value for each of the following functions is "AL", these tests always set EAX to zero or one. Therefore, you may refer to the AL or EAX register after these tests, whichever is more convenient for you. If you use instruction composition and bury one of these function calls in another statement, that statement will use the AL register as the operand.

Note that these functions generally pass their character set parameters by value. This involves pushing 16 bytes on the stack for each `cset` parameter (typically four push instructions). Keep this in mind if efficiency is your utmost concern. Be sure to read the section on "Passing CSET Parameters on the Stack" in the chapter on "Passing Parameters to Standard Library Routines".

```
procedure cs.IsEmpty( src: cset ); @returns( "AL" );
```

This function returns true (1) in the AL register if the specified character set is empty (has no members). It returns false (0) in AL/EAX otherwise.

HLA high-level calling sequence examples:

```
cs.IsEmpty( csetVar );
mov( al, booleanResult );
```

HLA low-level calling sequence examples:

```
// cs.IsEmpty is really intended to be used as a high-level
// type function. It's actually just as easy to compute the
// function manually as it is to call it. Here's the low-level
// calling sequence:

push( (type dword csetVar[12]));
push( (type dword csetVar[8]));
push( (type dword csetVar[4]));
```

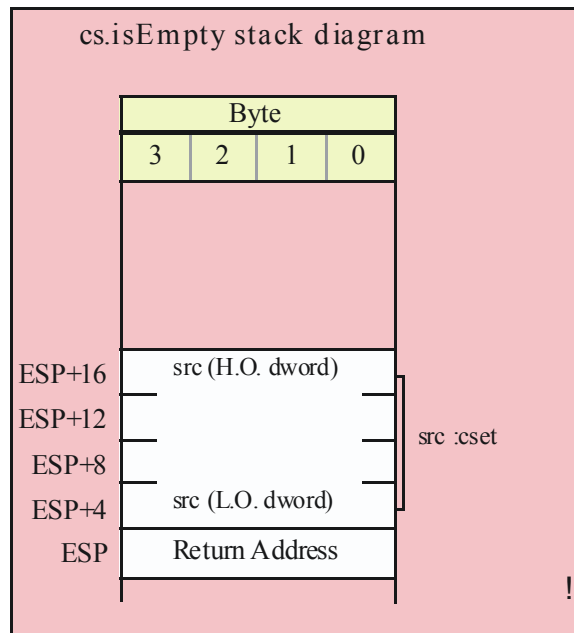
```

push( (type dword csetVar[0]));
call cs.IsEmpty;
mov( al, booleanResult;

// Here's the same thing using bare machine instructions:

mov( (type dword csetVar[0]), eax );
or( (type dword csetVar[4]), eax );
or( (type dword csetVar[8]), eax );
or( (type dword csetVar[12]), eax );
setz( al );
mov( al, booleanResult );

```



```

procedure cs.member( c:char; theSet:cset ); @returns( "AL" );

```

This function returns true (1) or false (0) in AL/EAX if the specified character is a member of the specified character set.

HLA high-level calling sequence examples:

```

cs.member( charVar, csetVar );
mov( al, booleanResult );

```

HLA low-level calling sequence examples:

```

// cs.member is really intended to be used as a high-level
// type function. It's actually just as easy to compute the
// function manually as it is to call it. Here's the low-level
// calling sequence:

movzx( charVar, eax );
push( eax );
push( (type dword csetVar[12]));
push( (type dword csetVar[8]));

```



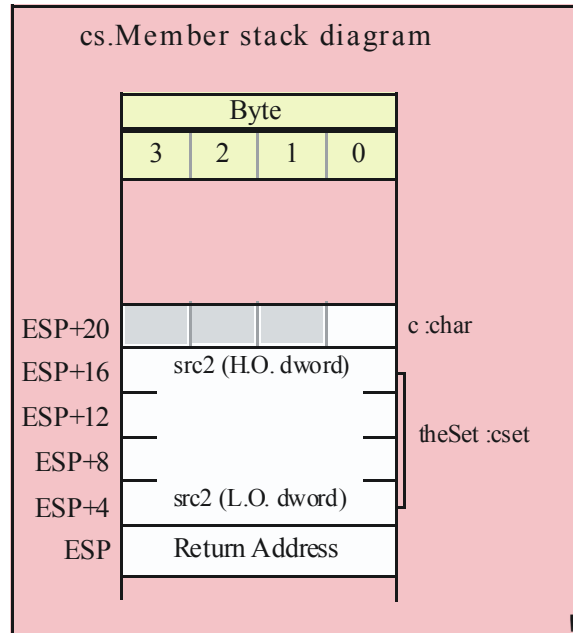
```

push( (type dword csetVar[4]));
push( (type dword csetVar[0]));
call cs.member;
mov( al, booleanResult;

// Here's the same thing using bare machine instructions:

movzx( charVar, eax );
bt( eax, csetVar );
setc( al );
mov( al, booleanResult );

```



```
procedure cs.subset( src1:cset; src2:cset ); @returns( "AL" );
```

The cs.subset function returns true in AL/EAX if src1 <= src2 (that is, all of src1's members are also members of src2).

HLA high-level calling sequence examples:

```

cs.subset( subsetVar, supersetVar );
mov( al, booleanResult );

```

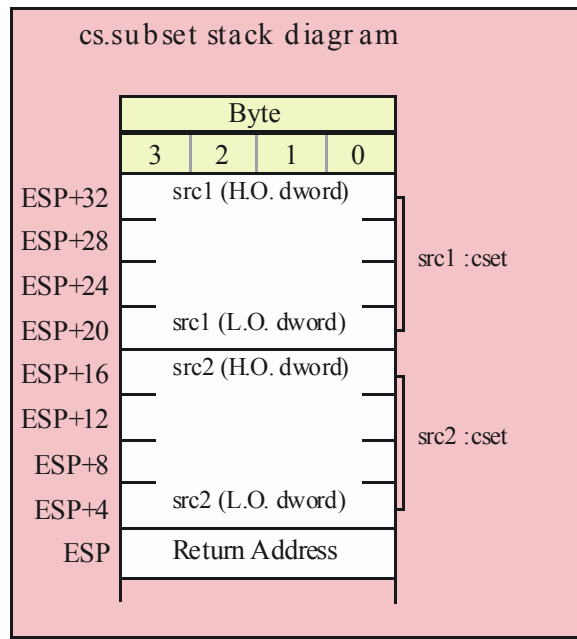
HLA low-level calling sequence examples:

```

push( (type dword subsetVar[12]));
push( (type dword subsetVar[8]));
push( (type dword subsetVar[4]));
push( (type dword subsetVar[0]));

push( (type dword supersetVar [12]));
push( (type dword supersetVar [8]));
push( (type dword supersetVar [4]));
push( (type dword supersetVar [0]));
call cs.subset;
mov( al, booleanResult );

```



```
procedure cs.superset( src1:cset; src2:cset ); @returns( "AL" );
```

The cs.superset function returns true in AL/EAX if src1 >= src2 (that is, all of src2's members are members of src1).

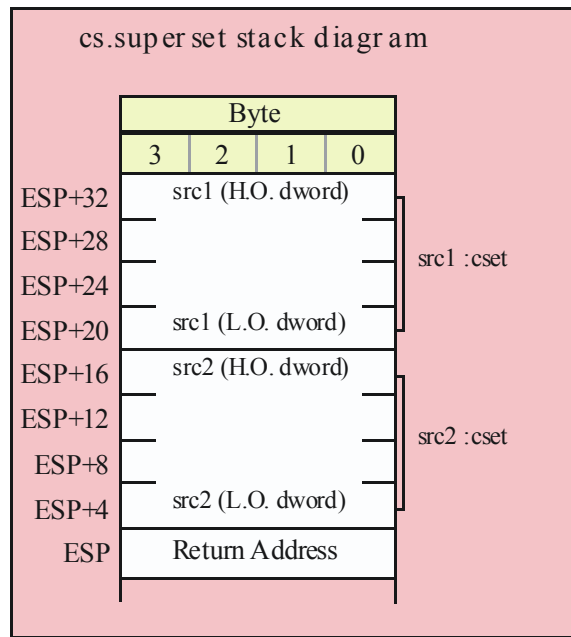
HLA high-level calling sequence examples:

```
cs.superset( supersetVar, subsetVar );
mov( al, booleanResult );
```

HLA low-level calling sequence examples:

```
push( (type dword supersetVar[12]));
push( (type dword supersetVar[8]));
push( (type dword supersetVar[4]));
push( (type dword supersetVar[0]));
```

```
push( (type dword subsetVar[12]));
push( (type dword subsetVar[8]));
push( (type dword subsetVar[4]));
push( (type dword subsetVar[0]));
call cs.superset;
mov( al, booleanResult );
```



```
procedure cs.psubset( src1:cset; src2:cset ); @returns( "AL" );
```

The cs.psubset (proper subset) function returns true in AL/EAX if  $\text{src1} < \text{src2}$  (that is, all of src1's members are members of src2 but  $\text{src1} \neq \text{src2}$ ).

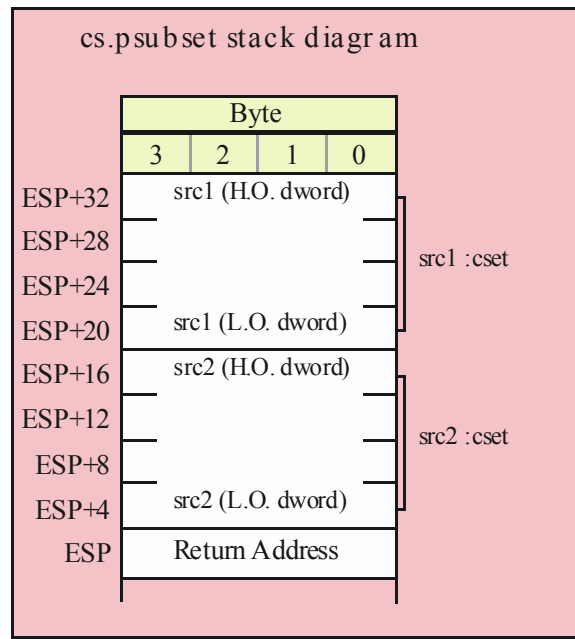
HLA high-level calling sequence examples:

```
cs.psubset( subsetVar, supersetVar );
mov( al, booleanResult );
```

HLA low-level calling sequence examples:

```
push( (type dword subsetVar[12]));
push( (type dword subsetVar[8]));
push( (type dword subsetVar[4]));
push( (type dword subsetVar[0]));

push( (type dword supersetVar [12]));
push( (type dword supersetVar [8]));
push( (type dword supersetVar [4]));
push( (type dword supersetVar [0]));
call cs.psubset;
mov( al, booleanResult );
```



```
procedure cs.psuperset( src1:cset; src2:cset ); @returns( "AL" );
```

The cs.psuperset (proper superset) function returns true in AL/EAX if src1 > src2 (that is, all of src2's members are members of src1 but src2 <> src1).

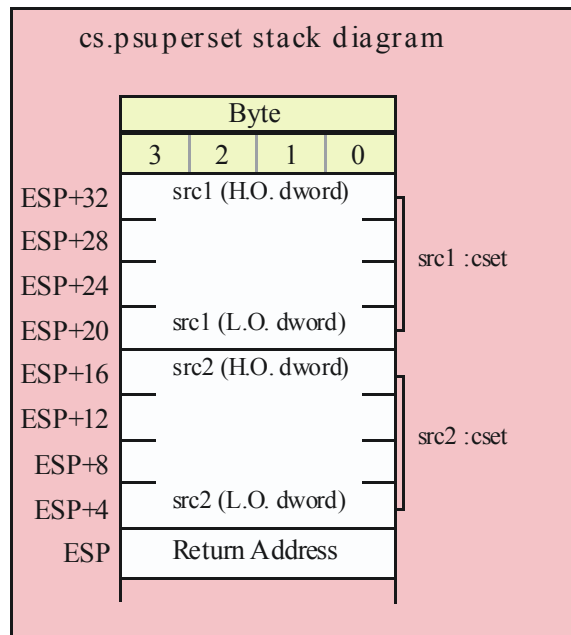
HLA high-level calling sequence examples:

```
cs.psuperset( supersetVar, subsetVar );
mov( al, booleanResult );
```

HLA low-level calling sequence examples:

```
push( (type dword supersetVar[12]));
push( (type dword supersetVar[8]));
push( (type dword supersetVar[4]));
push( (type dword supersetVar[0]));
```

```
push( (type dword subsetVar[12]));
push( (type dword subsetVar[8]));
push( (type dword subsetVar[4]));
push( (type dword subsetVar[0]));
call cs.psuperset;
mov( al, booleanResult );
```



```
procedure cs.eq( src1:cset; src2:cset ); @returns( "AL" );
```

The cs.eq function compares the two sets and returns true/false in AL/EAX; true if the two sets are equal, false if they are not.

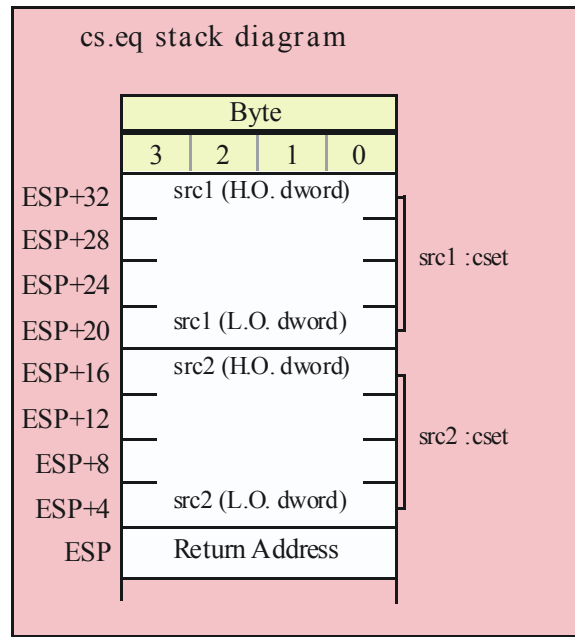
HLA high-level calling sequence examples:

```
cs.eq( src1, src2 );
mov( al, booleanResult );
```

HLA low-level calling sequence examples:

```
push( (type dword src1[12]));
push( (type dword src1[8]));
push( (type dword src1[4]));
push( (type dword src1[0]));

push( (type dword src2[12]));
push( (type dword src2[8]));
push( (type dword src2[4]));
push( (type dword src2[0]));
call cs.eq;
mov( al, booleanResult );
```



```
procedure cs.ne( src1:cset; src2:cset ); @returns( "AL" );
```

The cs.eq function compares the two sets and returns true/false in AL/EAX; true if the two sets are not equal, false if they are equal.

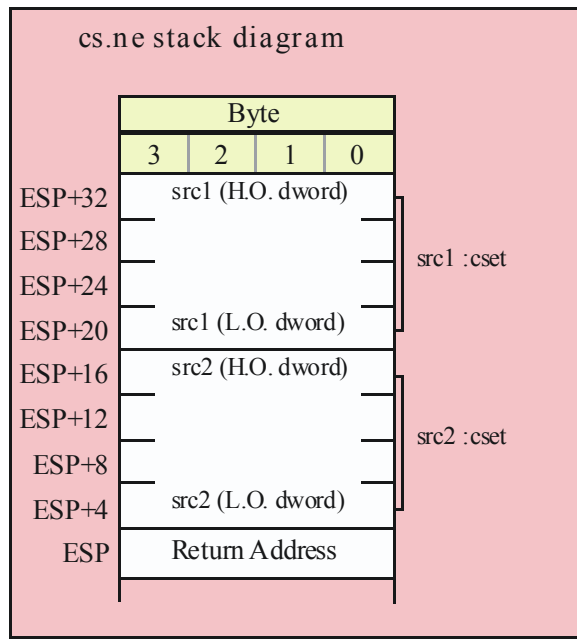
HLA high-level calling sequence examples:

```
cs.ne( src1, src2 );
mov( al, booleanResult );
```

HLA low-level calling sequence examples:

```
push( (type dword src1[12]));
push( (type dword src1[8]));
push( (type dword src1[4]));
push( (type dword src1[0]));

push( (type dword src2[12]));
push( (type dword src2[8]));
push( (type dword src2[4]));
push( (type dword src2[0]));
call cs.ne;
mov( al, booleanResult );
```



## 10.2 Character Set Construction and Manipulation

The functions in this group create character set objects, extract data from character set objects, or transfer data between character set objects.

**procedure cs.empty( var dest:cset );**

This function clears all the bits in a character set to create the empty set. Note that the single character set parameter is passed by reference.

HLA high-level calling sequence examples:

```
cs.empty( csetVar );
```

HLA low-level calling sequence examples:

```
// cset_s is a variable declared in the static/storage section:
```

```
pushd( &cset_s );
call cs.empty;
```

```
// cset_v is a variable declared in the var section or
// is a parameter:
```

```
lea( eax, cset_v );
push( eax );
call cs.empty;
```

```
// Alternative call passing cset_v if no 32-bit registers
// are available (this code assumes that EBP points at the current
// activation record/stack frame that contains cset_v):
```

```
push( ebp );
add( @offset( cset_v ), (type dword [esp] ));
call cs.empty;
```

```

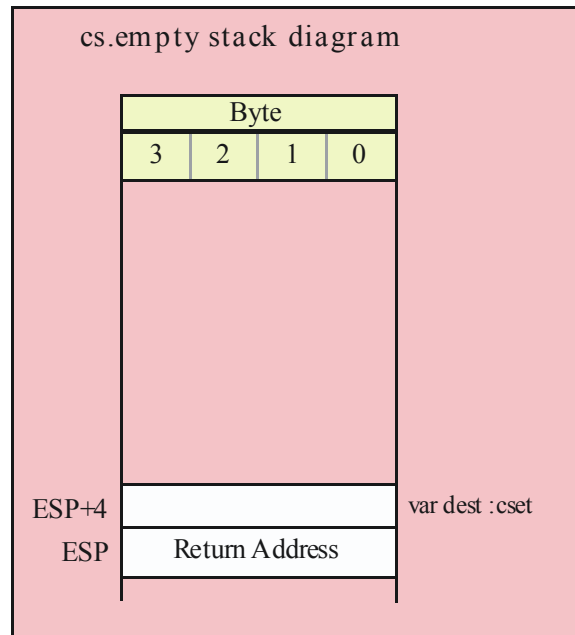
// Low-level call assuming a 32-bit register (esi in this case)
// contains the address of the cset:

push( esi );
call cs.empty;

// Low-level call assuming a dword or pointer variable contains the
// address of the cset that will receive the delimiter character set:

push( ptrToDelims );
call cs.empty;

```



```
procedure cs.cpy( src:cset; var dest:cset );
```

This routine copies the data from the source character set (src) to the destination character set (dest). Note that the dest set is passed by reference. Although this routine is convenient, you should consider writing a macro to do this same function (copy 16 bytes from src to dest) if you call this function in time critical sections of your code.

HLA high-level calling sequence examples:

```

cs.cpy( csetSrc, csetDest );
cs.cpy( {'a'..'z'}, lowerCaseCset );

```

HLA low-level calling sequence examples:

```

// csetDest_s is a variable declared
// in the static/storage section:

push( (type dword csetSrc_s[12]));
push( (type dword csetSrc_s[8]));
push( (type dword csetSrc_s[4]));
push( (type dword csetSrc_s[0]));

```



```

pushd( &csetDest_s );
call cs.cpy;

// csetDest_v is a variable declared in the var section or
// is a parameter:

push( (type dword csetSrc_v[12]));
push( (type dword csetSrc_v[8]));
push( (type dword csetSrc_v[4]));
push( (type dword csetSrc_v[0]));
lea( eax, csetDest_v );
push( eax );
call cs.cpy;

// Alternative call passing csetDest_v if no 32-bit registers
// are available (this code assumes that EBP points at the current
// activation record/stack frame that contains csetDest_v):

push( (type dword csetSrc_v[12]));
push( (type dword csetSrc_v[8]));
push( (type dword csetSrc_v[4]));
push( (type dword csetSrc_v[0]));
push( ebp );
add( @offset( csetDest_v ), (type dword [esp] ));
call cs.cpy;

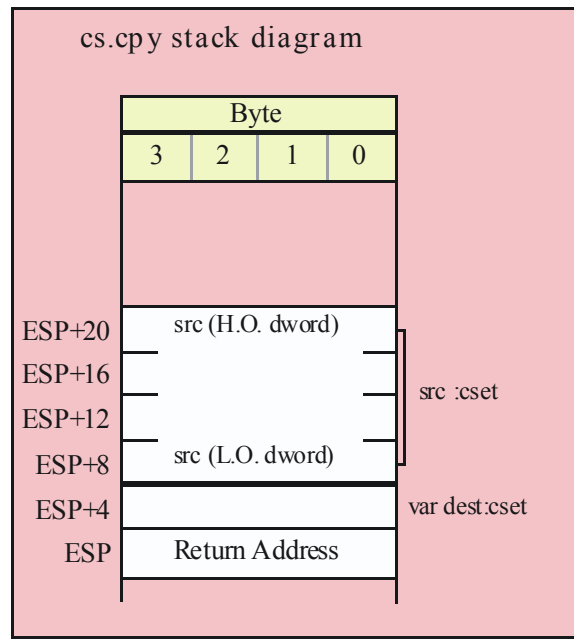
// Low-level call assuming a 32-bit register (edi in this case)
// contains the address of the destination cset:

push( (type dword csetSrc_v[12]));
push( (type dword csetSrc_v[8]));
push( (type dword csetSrc_v[4]));
push( (type dword csetSrc_v[0]));
push( edi );
call cs.cpy;

// Low-level call assuming a dword or pointer variable contains the
// address of the destination cset:

push( (type dword csetSrc[12]));
push( (type dword csetSrc[8]));
push( (type dword csetSrc[4]));
push( (type dword csetSrc[0]));
push( ptrToDest );
call cs.cpy;

```



```
procedure cs.charToCset( c:char; var dest:cset );
```

The `cs.charToCset` procedure takes the character passed as a parameter and creates a singleton set containing that character (a singleton is a set with exactly one member). The resulting set is stored into the destination parameter (which is passed by reference).

HLA high-level calling sequence examples:

```
cs.charToCset( 'c', csetDest );
cs.charToCset( charVar, lowerCaseCset );
cs.charToCset( (type char [esi]), (type cset [edi]));
```

HLA low-level calling sequence examples:

```
// The following low-level examples all assume that csetDest_s is
// a statically-declared object (static or storage section). For
// examples of additional ways to pass the destination cset by
// reference, please see the examples for cs.cpy given earlier.
//
// Passing a single character constant:
```

```
pushd( 'c' );
pushd( &csetDest_s );
call cs.charToCset;
```

```
// Passing a character variable, assuming a 32-bit register is
// available or one of the 8-bit register: AL, BL, CL, or DL
```

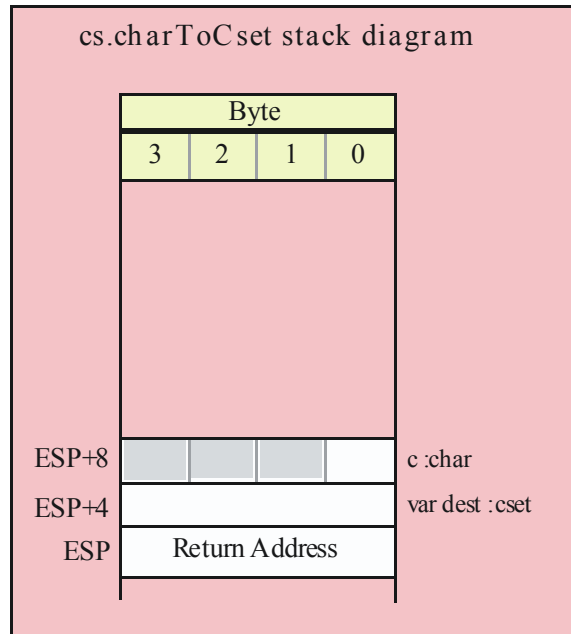
```
mov( charVar, al );
push( eax );
pushd( &csetDest_s );
call cs.charToCset;
```

```
// If the character variable is guaranteed not to be in the last
// three bytes of allocated storage, you could also do this:

push( type dword charVar );
pushd( &csetDest_s );
call cs.charToCset;

// If the character is in one of the 8-bit registers: AH, BH, CH, DH

sub( 4, esp );
mov( ah, [esp] );
pushd( &csetDest_s );
call cs.charToCset;
```



```
procedure cs.rangeChar( first:char; last:char; var dest:cset );
```

This function creates a set whose member range between the first character specified and the last character specified. For example, `cs.rangeChar( 'A', 'Z', UpperCaseSet )` will create a character set whose members are the upper case alphabetic characters. Any previous members in the destination set are lost.

HLA high-level calling sequence examples:

```
cs.rangeChar( 'a', 'z', csetDest );
cs.rangeChar( charVar, endCharVal, lowerCaseCset );
cs.rangeChar( (type char [esi]), '0', (type cset [edi]));
```

HLA low-level calling sequence examples:

```
// The following low-level examples all assume that csetDest_s is
// a statically-declared object (static or storage section). For
// examples of additional ways to pass the destination cset by
// reference, please see the examples for cs.cpy given earlier.
// Also note that both the "first" and "last" parameters are
// character objects that you pass the same way. For brevity,
// the following examples only demonstrate variations on the
```

```

// "first" parameter; the same principles apply to the "last"
// parameter.
//
// Passing two character constants:

pushd( 'a' );
pushd( 'z' );
pushd( &csetDest_s );
call cs.rangeChar;

// Passing a character variable, assuming a 32-bit register is
// available or one of the 8-bit register: AL, BL, CL, or DL

mov( charVar, al );
push( eax );
pushd( 'z' );
pushd( &csetDest_s );
call cs.rangeChar;

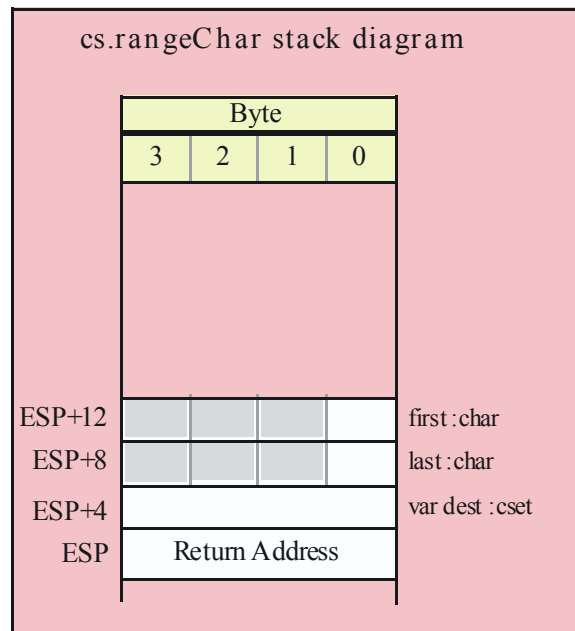
// If the "first" character variable is guaranteed not to be in the
// last three bytes of allocated storage, you could also do this:

push( type dword charVar );
pushd( 'z' );
pushd( &csetDest_s );
call cs.rangeChar;

// If the "first" character is in one of the 8-bit
// registers: AH, BH, CH, DH

sub( 4, esp );
mov( ah, [esp] );
pushd( 'z' );
pushd( &csetDest_s );
call cs.rangeChar;

```



**procedure cs.strToCset( s:string; var dest:cset );**

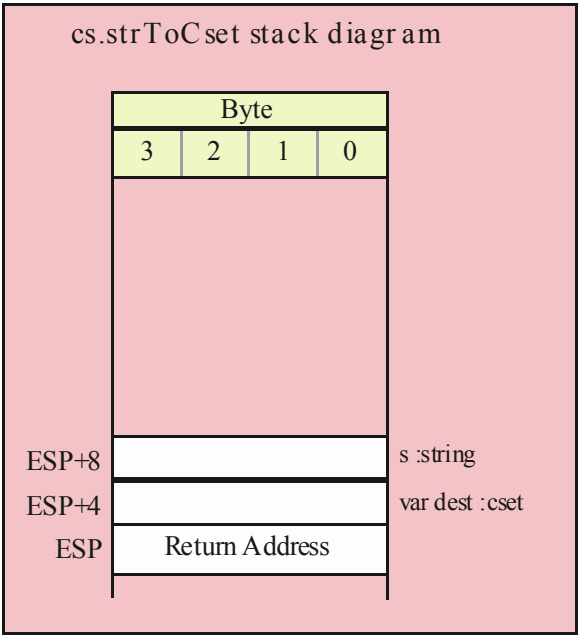
This function first sets the destination character set to the empty set. Then it "unions in" all the characters found in the string parameter to the destination set.

HLA high-level calling sequence examples:

```
cs.strToCset( strSrc, csetDest );  
cs.strToCset( "ABCDEFabcdef", hexCset );
```

HLA low-level calling sequence examples:

```
// csetDest_s is a variable declared  
// in the static/storage section:  
  
push( strSrc );  
pushd( &csetDest_s );  
call cs.strtoCset;  
  
// You could pass a string literal thusly (though there is  
// no benefit to doing this over creating a statically  
// initialized string variable and passing that string variable).  
  
lea( eax, "abcdefABCDEF" );  
push( eax );  
pushd( &csetDest_s );  
call cs.strtoCset;
```



**procedure cs.strToCset2( s:string; offs:uns32; var dest:cset );**

This function first sets the destination character set to the empty set. Then it "unions in" all the characters starting at offset `offs` in the string parameter to the destination character set.

HLA high-level calling sequence examples:

```
cs.strToCset2( strSrc, 2, csetDest );
cs.strToCset2( "ABCDEF", offsetIntoStr, partialHexCset );
```

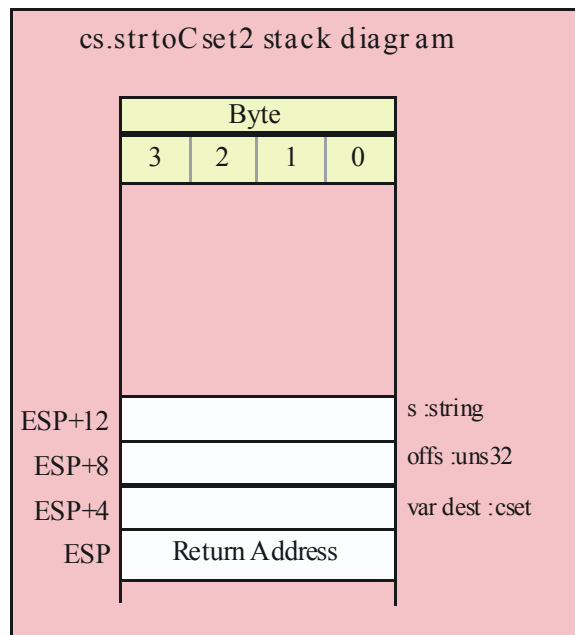
HLA low-level calling sequence examples:

```
// csetDest_s is a variable declared
// in the static/storage section:

push( strSrc );
pushd( 2 );
pushd( &csetDest_s );
call cs.strToCset2;

// Assume the offset is in the variable "offsetIntoStr":

push( strSrc );
push( offsetIntoStr );
pushd( &csetDest_s );
call cs.strToCset2;
```



```
procedure cs.extract( var dest:cset ); @returns( "EAX" );
```

This function removes a single character from the character set and returns that character in the AL register. Currently, this function removes characters by order of their ASCII character codes (that is, each call returns the character in the set with the lowest ASCII code). However, you should not make this assumption. You should assume that this function could return the characters in an arbitrary order. If the specified character set is empty, this routine returns -1 (\$FFFF\_FFFF) in the EAX register; in all other cases the H.O. three bytes of EAX contain zero upon return.

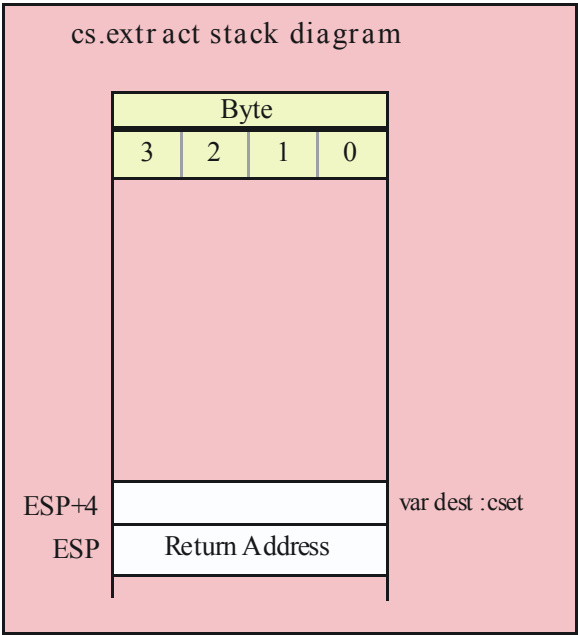
Note: unlike the HLA compile-time function "@extract", this function actually removes the character from the character set ("@extract" leaves the character in the set). Keep this in mind. (In the future, the name of the HLA @extract function will probably be changed to something else to clean up this conflict.)

HLA high-level calling sequence examples:

```
cs.extract( csetVar );  
if( eax <> -1 ) then  
  
    mov( al, charExtracted );  
  
endif;
```

HLA low-level calling sequence example:

```
// The following low-level example assumes that cset_s is  
// a statically-declared object (static or storage section). For  
// examples of additional ways to pass the destination cset by  
// reference, please see the examples for cs.cpy given earlier.  
  
pushd( &cset_s );  
call cs.extract;  
cmp( eax, -1 );  
je noCharExtracted;  
mov( al, charExtracted;  
noCharExtracted:
```



### 10.3 Set Operations

The following set functions perform what is generally considered to be set arithmetic: operations like set union, intersection, difference, and so on.

```
procedure cs.setunion( src:cset; var dest:cset );
```

This function computes the union of two sets, storing the result back into the destination set. Note that the destination set parameter is passed by reference.

Note: The name "setunion" was used rather than the more obvious choice of "union" because "union" is an HLA reserved word.

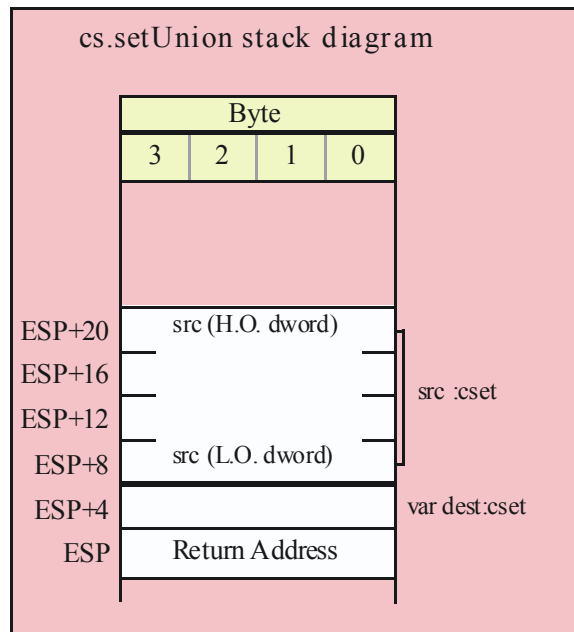
HLA high-level calling sequence examples:

```
cs.setunion( csetSrc, csetDest );
cs.setunion( {'a'..'z'}, lowerCaseUnion );
```

HLA low-level calling sequence example:

```
// The following low-level example assumes that csetDest_s is
// a statically-declared object (static or storage section). For
// examples of additional ways to pass the destination cset by
// reference, please see the examples for cs.cpy given earlier.
```

```
push( (type dword csetSrc_s[12]));
push( (type dword csetSrc_s[8]));
push( (type dword csetSrc_s[4]));
push( (type dword csetSrc_s[0]));
pushd( &csetDest_s );
call cs.setunion;
```



```
procedure cs.intersection( src:cset; var dest:cset );
```

This function computes the set intersection of the two sets passed as parameters and stores the result back into the destination set. Note that the dest parameter is passed by reference.

HLA high-level calling sequence examples:

```
cs.intersection( csetSrc, csetDest );
```

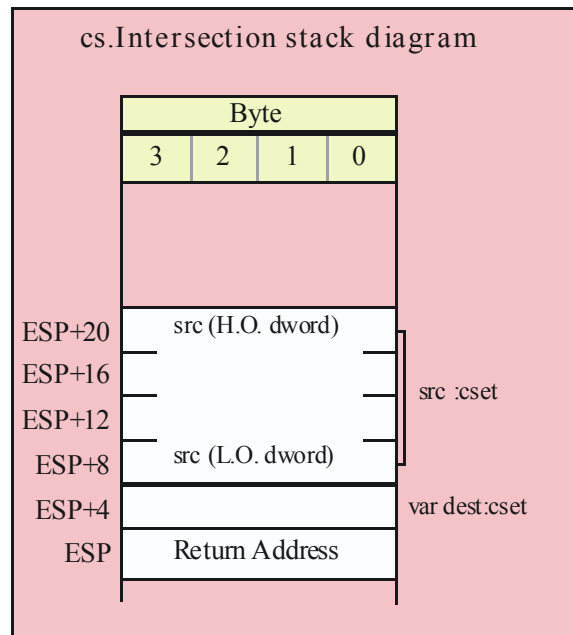


```
cs.intersection( {'a'..'z'}, lowerCaseUnion );
```

HLA low-level calling sequence example:

```
// The following low-level example assumes that csetDest_s is
// a statically-declared object (static or storage section). For
// examples of additional ways to pass the destination cset by
// reference, please see the examples for cs.cpy given earlier.
```

```
push( (type dword csetSrc_s[12]));
push( (type dword csetSrc_s[8]));
push( (type dword csetSrc_s[4]));
push( (type dword csetSrc_s[0]));
pushd( &csetDest_s );
call cs.intersection;
```



```
procedure cs.difference( src:cset; var dest:cset );
```

This function computes the set difference of two sets (i.e., the members in the destination set that are not also members of the source set). It stores the result back into the dest set (which is passed by reference).

HLA high-level calling sequence examples:

```
cs.difference( csetSrc, csetDest );
cs.difference( {'a'..'z'}, csetWithoutLowerCase );
```

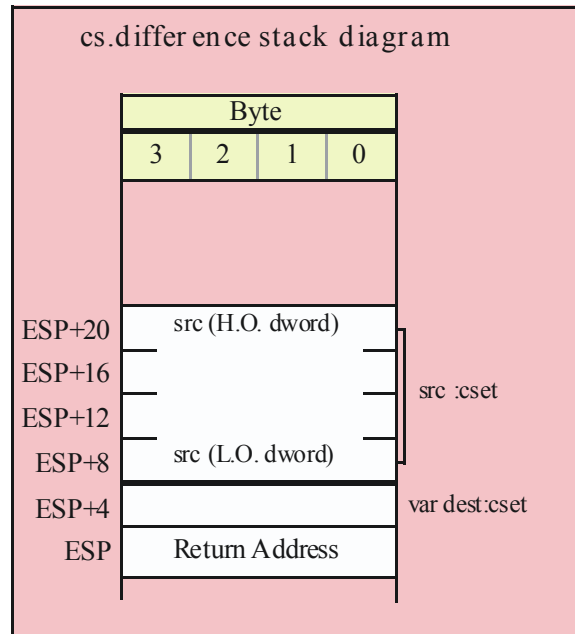
HLA low-level calling sequence example:

```
// The following low-level example assumes that csetDest_s is
// a statically-declared object (static or storage section). For
// examples of additional ways to pass the destination cset by
// reference, please see the examples for cs.cpy given earlier.
```

```

push( (type dword csetSrc_s[12]));
push( (type dword csetSrc_s[8]));
push( (type dword csetSrc_s[4]));
push( (type dword csetSrc_s[0]));
pushd( &csetDest_s );
call cs.difference;

```



```

procedure cs.complement( src:cset; var dest:cset );

```

This function computes the set complement of a set (i.e., the members in the destination set are those elements that are not in the source set.). It stores the complemented version of the set in the destination operand (which is passed by reference).

HLA high-level calling sequence examples:

```

cs.complement( csetSrc, negatedCsetDest );
cs.complement( {'a'..'z'}, allButLowercase );

```

HLA low-level calling sequence example:

```

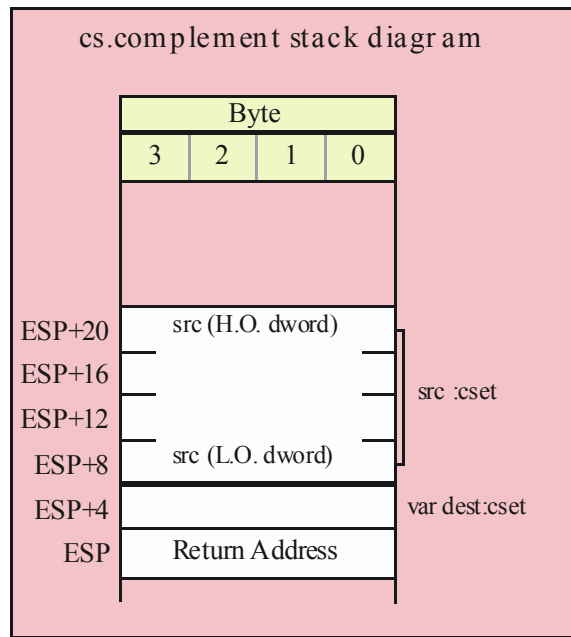
// The following low-level example assumes that csetDest_s is
// a statically-declared object (static or storage section). For
// examples of additional ways to pass the destination cset by
// reference, please see the examples for cs.cpy given earlier.

```

```

push( (type dword csetSrc_s[12]));
push( (type dword csetSrc_s[8]));
push( (type dword csetSrc_s[4]));
push( (type dword csetSrc_s[0]));
pushd( &csetDest_s );
call cs.complement;

```



```
procedure cs.unionChar( c:char; var dest:cset );
```

The cs.unionChar function adds the character (supplied as a parameter) to the specified destination character set (passed by reference). If the character was already a member of the set, this function does not affect the character set.

HLA high-level calling sequence examples:

```
cs.unionChar( 'c', csetDest );
cs.unionChar( charVar, lowerCaseCset );
cs.unionChar( (type char [esi]), (type cset [edi]));
```

HLA low-level calling sequence examples:

```
// The following low-level examples all assume that csetDest_s is
// a statically-declared object (static or storage section). For
// examples of additional ways to pass the destination cset by
// reference, please see the examples for cs.cpy given earlier.
//
// Passing a single character constant:

pushd( 'c' );
pushd( &csetDest_s );
call cs.unionChar;

// Passing a character variable, assuming a 32-bit register is
// available or one of the 8-bit register: AL, BL, CL, or DL

mov( charVar, al );
push( eax );
pushd( &csetDest_s );
call cs.unionChar;
```

```

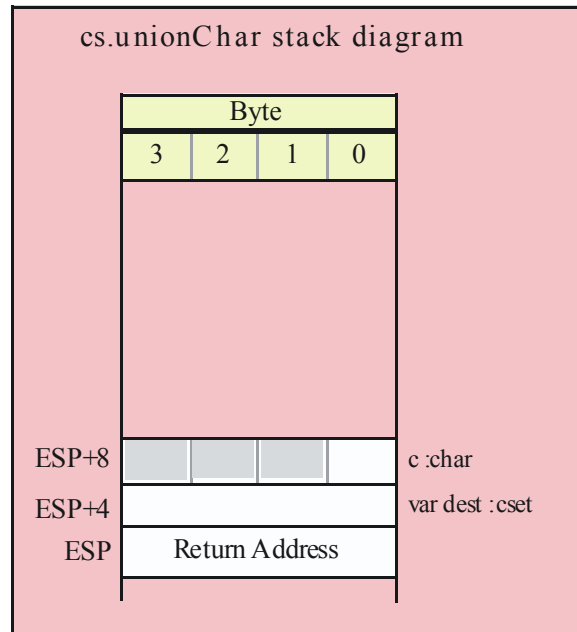
// If the character variable is guaranteed not to be in the last
// three bytes of allocated storage, you could also do this:

push( type dword charVar );
pushd( &csetDest_s );
call cs.unionChar;

// If the character is in one of the 8-bit registers: AH, BH, CH, DH

sub( 4, esp );
mov( ah, [esp] );
pushd( &csetDest_s );
call cs.unionChar;

```



```
procedure cs.removeChar( c:char; var dest:cset );
```

This function removes a single character from the specified destination set (passed by reference). If the character was not previously a member of the destination set, this function does not affect that set.

HLA high-level calling sequence examples:

```

cs.removeChar( 'c', csetDest );
cs.removeChar( charVar, lowerCaseCset );
cs.removeChar( (type char [esi]), (type cset [edi]));

```

HLA low-level calling sequence examples:

```

// The following low-level examples all assume that csetDest_s is
// a statically-declared object (static or storage section). For
// examples of additional ways to pass the destination cset by
// reference, please see the examples for cs.cpy given earlier.
//

```

```
// Passing a single character constant:

pushd( 'c' );
pushd( &csetDest_s );
call cs.removeChar;

// Passing a character variable, assuming a 32-bit register is
// available or one of the 8-bit register: AL, BL, CL, or DL

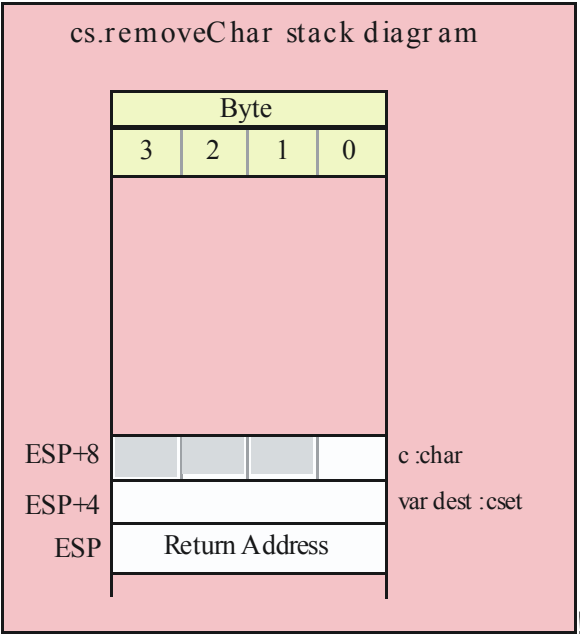
mov( charVar, al );
push( eax );
pushd( &csetDest_s );
call cs.removeChar;

// If the character variable is guaranteed not to be in the last
// three bytes of allocated storage, you could also do this:

push( type dword charVar );
pushd( &csetDest_s );
call cs.removeChar;

// If the character is in one of the 8-bit registers: AH, BH, CH, DH

sub( 4, esp );
mov( ah, [esp] );
pushd( &csetDest_s );
call cs.removeChar;
```



```
procedure cs.unionStr( s:string; var dest:cset );
```

This function will union in all the characters in a string to the destination set. Unlike the `cs.strToCset` function, this function does not clear the destination character set before processing the characters in the string.

HLA high-level calling sequence examples:

```
cs.unionStr( strSrc, csetDest );
cs.unionStr( "ABCDEFabcdef", csetPlusHexChars );
```

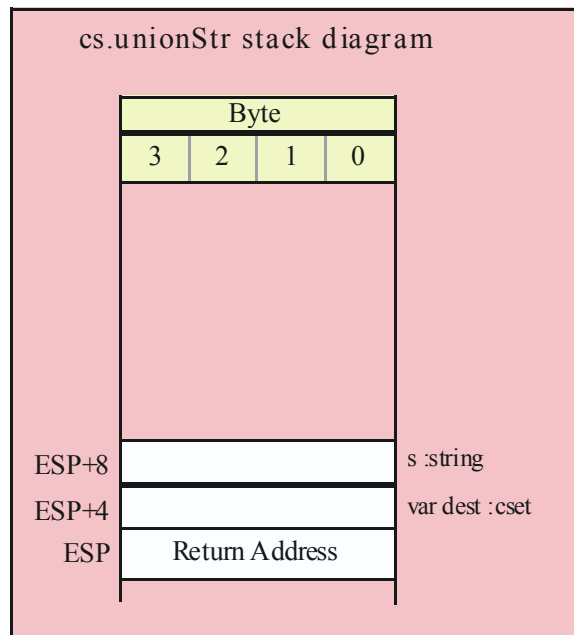
HLA low-level calling sequence examples:

```
// csetDest_s is a variable declared
// in the static/storage section:
```

```
push( strSrc );
pushd( &csetDest_s );
call cs.unionStr;
```

```
// You could pass a string literal thusly (though there is
// no benefit to doing this over creating a statically
// initialized string variable and passing that string variable).
```

```
lea( eax, "abcdefABCDEF" );
push( eax );
pushd( &csetDest_s );
call cs.unionStr;
```



```
procedure cs.unionStr2( s:string; offs:uns32; offs:uns32; var dest:cset );
```

This function will union in all the characters in a string to the destination set. Unlike the cs.unionStr function, this function starts at character position offs in s rather than at character position zero.

HLA high-level calling sequence examples:

```
cs.unionStr2( strSrc, 2, csetDest );
cs.unionStr2( "ABCDEF", offsetIntoStr, partialHexUnion );
```

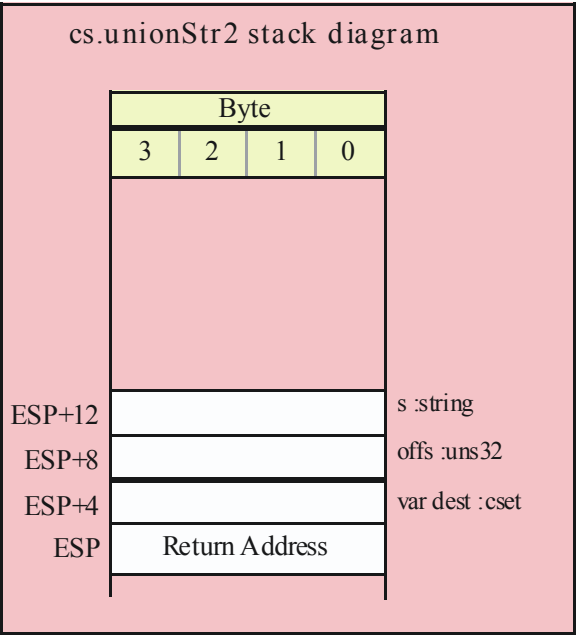
HLA low-level calling sequence examples:

```
// csetDest_s is a variable declared
// in the static/storage section:

push( strSrc );
pushd( 2 )
pushd( &csetDest_s );
call cs.unionStr2;

// Assume the offset is in the variable "offsetIntoStr":

push( strSrc );
push( offsetIntoStr );
pushd( &csetDest_s );
call cs.unionStr2;
```



**procedure cs.removeStr( s:string; var dest:cset );**

This function removes characters found in the string from the specified character set. If a character in the string was not previously a member of the character set, the specified character has no effect on the destination set.

HLA high-level calling sequence examples:

```
cs.removeStr( strSrc, csetDest );
cs.removeStr( "ABCDEFabcdef", csetMinusHexChars );
```

HLA low-level calling sequence examples:

```
// csetDest_s is a variable declared
// in the static/storage section:
```

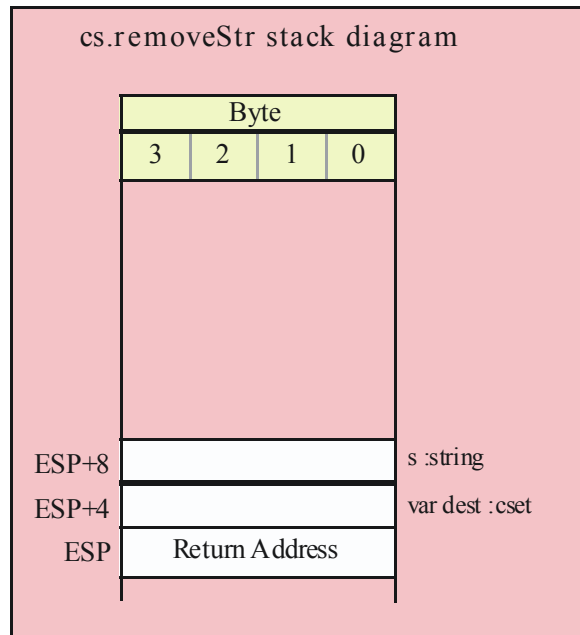
```

push( strSrc );
pushd( &csetDest_s );
call cs.removeStr;

// You could pass a string literal thusly (though there is
// no benefit to doing this over creating a statically
// initialized string variable and passing that string variable).

lea( eax, "abcdefABCDEF" );
push( eax );
pushd( &csetDest_s );
call cs.removeStr;

```



```
procedure cs.removeStr2( s:string; offs:uns32; var dest:cset );
```

This function removes characters found in the string at character position offs and beyond from the specified character set. If a character in the string was not previously a member of the character set, the specified character has no effect on the destination set.

HLA high-level calling sequence examples:

```

cs.removeStr2( strSrc, 2, csetDest );
cs.removeStr2( "ABCDEF", offsetIntoStr, csetMinusSomeHexChars );

```

HLA low-level calling sequence examples:

```

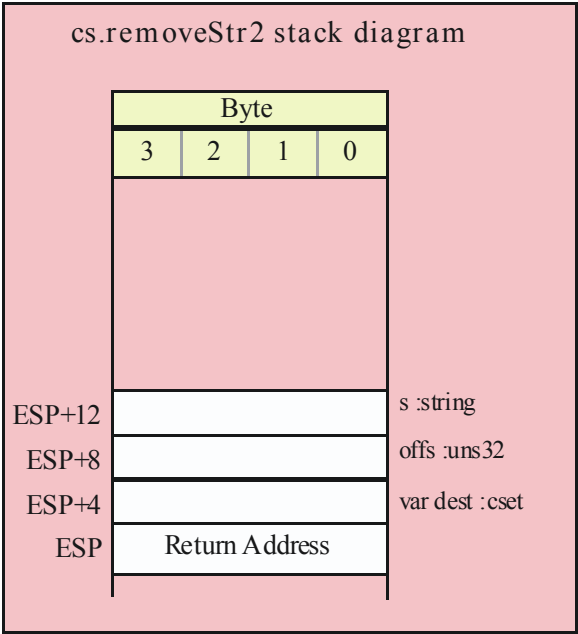
// csetDest_s is a variable declared
// in the static/storage section:

push( strSrc );
pushd( 2 );
pushd( &csetDest_s );
call cs.removeStr2;

```



```
// Assume the offset is in the variable "offsetIntoStr":  
  
push( strSrc );  
push( offsetIntoStr );  
pushd( &csetDest_s );  
call cs.removeStr2;
```





# 11 Date Functions (datetime.hhf)

HLA contains a set of procedures and functions that simplify *correct* date calculations. There are actually two modules: a traditional set of date functions and, for those who prefer an object-oriented approach, a date class module.

Note: be sure to read the chapter on "Passing Parameters to Standard Library Routines" (parmpassing.rtf) before reading this chapter.

**A Note About Thread Safety:** The date and time routines maintain a couple of static global variables that track the output format and output separate characters for dates. Currently, these values apply to all threads in a process. You should take care when changing these values in threads. When the process module is added to the standard library, these values will be placed in a per-thread data structure. Until then, you should set the format/separator character before starting any other threads and avoid changing their values once other threads (that might use the date/time library module) begin execution.

**Note about stack diagrams:** this documentation includes stack diagrams for those functions that pass parameters on the stack. To conserve space, this documentation does not include a stack diagram for any function that does not pass data on the stack (that is, only a return address appears on the stack).

**Note about function overloading:** the functions in the date/time module use function overloading in order to allow you to specify the parameter lists in different ways. The macro that handles the overloading generally coerces the possible parameter types into a single object that it passes to the underlying function. The documentation for the specific functions will tell you whether a symbol is a macro or a function. For the most part, this should matter to you unless you are taking the address of a function (which you cannot do with a macro). See the HLA documentation for more details on function overloading via macros.

## 11.1 The Date Module

To use the date functions in your application, you will need to include one of the following statements at the beginning of your HLA application:

```
#include( "datetime.hhf" )
or
#include( "stdlib.hhf" )
```

## 11.2 Date Data Types

The date namespace defines the following useful data types:

### date.daterec

Date representation. This is a dword object containing *m*, *d*, and *y* fields holding the obvious values. The *y* field is a 16-bit quantity supporting years 0..9,999. No Y2K problems here! (Of course, it does suffer from Y10K, but that's probably okay.) Since the Gregorian calendar began use in Oct, 1582, there is really no need to represent dates any earlier than this. In fact, most date calculations in the HLA stdlib will not allow dates earlier than Jan 1, 1600 for this very reason. The limitation of year 9999 is an arbitrary limit set in the library to help catch wild values. If you really need dates beyond 9999, feel free to modify the date validation code. The *m* and *d* fields are both byte objects. The date validation routines enforce the month limits of 1..12 and appropriate day limits (depending on the month and year).

Here is the current data type definition for the *daterec* data type:

```
type
  daterec:
    record
      day      :uns8;
      month    :uns8;
      year     :uns16;
    endrecord;
```

Because of the way the fields are defined, you may compare two dates as 32-bit values and test the result using unsigned conditional branch instructions.

**date.outputFormat**

This is an enumerated data type that defines the following constants:

mdyy, mdyyyy, mmddyy, mmddyyyy, yyymd, yyyyymd, yymmdd, yyyyymmdd, MONdyyyy, and MONTHdyyyy. These constants control the date output format in the (mostly) obvious way. Note that mdyy can output one digit for the day and month while mmddyy always inputs two digits for each field. The MONdyyyy format outputs dates in the form "Jan 1, 2000" while the MONTHdyyyy outputs the dates using the format "January 1, 2000".

```
type
    OutputFormat:
        enum
        {

            mdyy,
            mdyyyy,

            mmddyy,
            mmddyyyy,

            yyymd,
            yyyyymd,
            yymmdd,
            yyyyymmdd,

            MONdyyyy,
            MONTHdyyyy,
            badDateFormat
        };
```

## 11.3 Date Tables

The date/time module includes several date/time-related data objects that may be of interest to an application programmer. Here are the declarations found in the datetime.hhf header file:

```
DaysToMonth      :uns32[13];
DaysInMonth      :uns32[13];
DaysFromMonth    :uns32[13];
Months           :string[13];
shortMonths      :string[13];
```

You must treat these tables as read-only objects. Changing their values will cause the date/time routines to produce incorrect results. Each of these tables is indexed by a month value in the range 1..12. Zero is an illegal value and the value found at index 0 in these tables is undefined. Obviously, accessing any data beyond index 12 is also undefined. The first three functions return some number of days relative to the month whose index you've supplied. These day values are relative to the first day of the specified month. The values in these tables are for non-leap years. If your date calculation is for a leap year, you must add one to the value found in these tables, as appropriate for the month you specify; details appear in the discussion of each function.

DaysToMonth contains the number of days from January 1 to the first of the month you specify as the index. For example, index 1 contains zero, index 2 contains 31, index 3 contains 59 (31+28), etc. For leap years, you will need to add one to the table entry if the index is in the range 3..12.

DaysInMonth contains the number of days in the month specified by the index. For example, DaysInMonth[1] contains 31, DaysInMonth[2] contains 28, and DaysInMonth[3] contains 31. For leap years, you need to add one to the value appearing at index 2 (of course, it's probably just easier to explicitly set the value to 29 for February in leap years).

DaysFromMonth contains the number of days from the first day of the month specified by the index to the first day of January in the following year. For example, DaysFromMonth[1] will contain 365, DaysFromMonth[2] will contain 334 (365-31), and so on. For leap years, you will want to add one to the value if the month index is less than 3.

Months is an array of strings, indexed by the month value, that contain the month's name. For example, Month[1] is the string "January" and Month[2] is the string "February".

shortMonths is an array of strings that contain shortened versions of the month names (the first three characters of each of the month names found in the Months array).

## 11.4 Date Predicates

The date module provides many functions that test date values. This section details those functions.

```
#macro date.isLeapYear( y:uns32 ); @returns( "al" );
#macro date.isLeapYear( dr:date.daterec ); @returns( "al" );
procedure date._isLeapYear( Year:word ); @returns( "al" );
```

This is an overloaded function. You may either pass it an unsigned integer containing the year or a date.daterec value specifying a m/d/y value (the overloading function will simply pick out the year value and pass it on to the underlying date.\_isLeapYear function). These functions return true or false in the AL register depending upon whether the parameter is a leap year (true if it is a leap year). Note that this function will be correct until sometime between the years 3000 and 4000, at which point people will probably have to agree upon adding an extra leap day in at some point (no such agreement has been made today, hence the absence from this function); currently, HLA date routines do not allow dates beyond the year 2999, so this won't be a problem unless you modify the maximum year value in the date/time header files. Note that these functions return the boolean result zero-extended into EAX. The @returns("al") declarations exist so that these functions will be type-compatible with boolean objects.

HLA high-level calling sequence examples:

```
date.isLeapYear( someDateVar );
mov( al, someDateVar_is_leap_year );
if( date.isLeapYear( aYearValue ) ) then

    // Do something if aYearValue is a leap year

endif;

mov( &date._isLeapYear( ebx ), ptrToIsLeapYearFunction );
```

HLA low-level calling sequence examples:

```
movzx( someDateVar.year, eax );
push( eax );
call date._isLeapYear;
mov( al, someDateVar_is_leap_year );

movzx( aYearValue, eax );
push( eax );
call date._isLeapYear;
test( al, al );
jz notALeapYear;

// Do something if aYearValue is a leap year

notALeapYear:
```

```

#macro date.validate( m:byte; day:byte; year:word );
#macro date.validate( dr:date.daterec );
date._validate( dr:daterec );

```

These two functions check the date passed as a parameter and raise an `ex.InvalidDate` exception if the data in the fields (or the m/d/y) parameter is not a valid date between 1/1/1600 and 12/31/2999.

HLA high-level calling sequence examples:

```

try

    date.validate( someDateVar );

    anyexception

    // Do something if the date is invalid

endtry;

try

    date.validate( someMonth, someDay, someYear );

    anyexception

    // Do something if the date is invalid

endtry;

try

    date._validate( someDateVar );

    anyexception

    // Do something if the date is invalid

endtry;

```

HLA low-level calling sequence examples:

```

push( someDateVar.date );
call date._validate;

```

```

#macro date.isValid( m:byte; day:byte; year:word ); @returns( "al" );
#macro date.isValid( dr:date.daterec ); @returns( "al" );
date._isValid( dr:daterec ); @returns( "al" );

```

Similar to the `date.validate` procedures, except these functions return true/false in the AL register if the date is valid/invalid. They do not raise an exception. Note that these functions return the boolean result zero-extended into EAX. The `@returns("al")` declarations exist so that these functions will be type-compatible with boolean objects.

## 11.5 Date Conversions

The functions (and macros) in this category convert dates from one format to another. Specifically, there are conversions to and from Julian day numbers, conversions to days into year, computation of week day (Sunday through Saturday), and conversions to days left in year.

```
#macro date.pack( m, d, y, dr );
```

*date.pack* is a macro that accepts year (*y*), month (*m*), and day (*d*) values (presumably dwords), and a *date.daterec* (*dr*) object. It converts the three values to *date.daterec* form and stores the result in the specified destination. If the *y*, *m*, or *d* values are constants, this macro checks them to ensure they are somewhat reasonable (days are only checked for the range 1..31, years are checked for the range 1600..2999). If *m* or *d* are memory objects, then they are coerced to a byte before use. If *y* is a memory object, it is coerced to a word before use. You may use registers for *y*, *m*, and *d*; if you do, the *m* and *d* values must be passed in 8-bit registers and the *y* value must be passed in a 16-bit register. This macro works best if all three operands are constants. If you take a look at the macro definition in the *datetime.hhf* header file, you'll discover that this macro efficiently translates a constant date into a single machine instruction. The macro attempts to generate good code for other operand types, but if efficiency is your primary concern, you may want to consider manually moving the data into the fields of the *daterec* object if the *d*, *m*, and *y* values are memory operands.

Because this is a macro, there are no parameters passed on the stack (hence, no stack diagram). Do note, however, that this macro preserves the value in EAX on the stack if it needs to use EAX. As such you should not specify an ESP-relative memory operand as one of the parameters to this macro. In some cases the macro will push EAX on the stack during conversion, in other cases it will not. As such, ESP-relative memory addresses may be rendered incorrect when this macro preserves EAX on the stack.

Other than a mild check for constant operands, this macro does not validate the date you pack into the *dr* argument. No checking is done because this macro is primarily intended for moving constant values into a *daterec* object (and you should be able to verify the value manually when writing the macro invocation). If you need to verify the date packed into the *dr* parameter, use the *date.validate* or *date.isValid* functions.

Macro invocation example:

```
date.pack( 6, 21, 2007, drDateVar );
```

```
#macro date.unpack( dr, m, d, y );
```

*date.unpack* is a macro that accepts a *daterec* object (*dr*) and converts this to three dword values (*m*, *d*, and *y*) by zero-extending the values before storing them into the destination locations. The *m*, *d*, and *y* operands must be 32-bit memory locations or registers (except EAX, which this macro uses for the zero extension operation).

Macro invocation example:

```
date.unpack( drDateVar, monthVar32, dayVar32, yearVar32 );
```

```
#macro date.toJulian( m:byte; d:byte; y:word ); @returns( "eax" );
```

```
#macro date.toJulian( dr:date.daterec ); @returns( "eax" );
```

```
date._toJulian( dr:daterec ); @returns( "eax" );
```

These functions convert the Gregorian (i.e., standard) date passed as a parameter into a "Julian day number." A Julian day number treats Jan 1, 4713 as day zero and simply numbers dates forward from that point. For example, Oct 9, 1995 is JD 2,450,000. Jan 1, 2000 is JD 2,452,545. Julian dates make date calculations of the form "what date is it X days from now?" trivial. You just compute JD+X to get the new date.

A Julian Day begins at 12:00 noon (compared with Gregorian days, that begin at 12:00 midnight). Because these functions do not have a time parameter, they assume that the time is between 12:00 noon on the specified date you pass as a parameter and 11:59:59 of the next day. If the current time is before 12:00 noon, you should subtract one from the Julian day number these functions return. If you would like a true 'toJulian' function, you can easily write one thusly:

```
procedure toJulian( dr:date.daterec; tm:time.timerec );
begin toJulian;
```

```

date.toJulian( dr );
if( tm.hour < 12 ) then
    dec( eax );
endif;

end toJulian;

```

`date.toJulian` is actual a macro that handles the parameter overloading. It reformats the parameters (as necessary) and calls the `date._toJulian` function to do the actual work. You would not normally call the `date._toJulian` function as the `date.toJulian` macro with a single argument makes this call for you. You would normally use the `date._toJulian` function in your applications if you need to pass the address of a function as a parameter to some other function (you cannot take the address of a macro).

Note: a "Julian Date" is not the same thing as a "Julian Day Number". A Julian date is based on the Julian Calendar created by Julius Caesar in about 45 BC. It was very similar to our current calendar except they didn't get the leap years quite right. Julian Day numbers are a different calendar system that, as explained above, number days consecutively after Jan 1 4713 BC (resetting to day one 7980 years later). Out of sheer laziness, this document will use the term "Julian Date" as a description of the calendar based on Julian day numbers despite that fact that this is technically incorrect.

HLA high-level calling sequence examples:

```

date.toJulian( someDateVar );
mov( eax, JulianDayNumber );
date.toJulian( month, day, year );
mov( eax, JulianDayNumber2 );

```

HLA low-level calling sequence examples:

```

push( someDateVar.date );
call date._toJulian;
mov( eax, JulianDayNumber );

```

**`date.fromJulian( jd:uns32; var gd:date.daterec );`**

This procedure converts a Julian date to a Gregorian date. The Julian date is the first parameter, the second (reference) parameter is a Gregorian date variable (`data.daterec`). See the note above about adding some number of days to a Gregorian date via translation to Julian. Note that adding months or years to a Julian date is a real pain in the rear. It's generally easier and faster to convert the Julian date to a Gregorian date, add the months and/or years to the Gregorian date (which is relatively easy), and then convert the whole thing back to a Julian day number.

HLA high-level calling sequence examples:

```

date.fromJulian( JulianDayNumber, someDateVar );

```

HLA low-level calling sequence examples:

```

// Assume someDateVar is a static variable:

push( JulianDayNumber );
pushd( &someDateVar );
call date.fromJulian;

// Assume someDateVar is not a static object

push( JulianDayNumber );
lea( eax, someDateVar );

```



```
push( eax );
call date.fromJulian;
```

```
#macro date.dayNumber( m:byte; d:byte; y:word ); @returns( "eax" );
#macro date.dayNumber( dr:date.daterec ); @returns( "eax" );
date._dayNumber( dr:daterec ); @returns( "eax" );
```

These functions convert the Gregorian date passed as a parameter into a day number into the current year (often erroneously called a "Julian Date" since NASA adopted this terminology in the late sixties). These functions return a value between 1 and 365 or 366 (for leap years) in the EAX register. Jan 1 is day 1, Dec 31 is day 365 or day 366.

HLA high-level calling sequence examples:

```
date.dayNumber( someDateVar );
mov( eax, dayNumber1 );
date.dayNumber( month, day, year );
mov( eax, dayNumber2 );
```

HLA low-level calling sequence examples:

```
push( someDateVar.date );
call date._dayNumber;
mov( eax, dayNumber1 );
```

```
#macro date.daysLeft( m:byte; d:byte; y:word ); @returns( "eax" );
#macro date.daysLeft( dr:date.daterec ); @returns( "eax" );
date._daysLeft( dr:daterec ); @returns( "eax" );
```

These functions return the number of days left in the current year *counting the date passed as a parameter* (hence Dec 31, yyyy always returns one).

HLA high-level calling sequence examples:

```
date.daysLeft( someDateVar );
mov( eax, daysLeft1 );
date.daysLeft( month, day, year );
mov( eax, daysLeft2 );
```

HLA low-level calling sequence examples:

```
push( someDateVar.date );
call date._daysLeft;
mov( eax, daysLeft1 );
```

```
#macro date.dayOfWeek( m:byte; d:byte; y:word ); @returns( "eax" );
#macro date.dayOfWeek( dr:date.daterec ); @returns( "eax" );
date._dayOfWeek( dr:daterec ); @returns( "eax" );
```

These functions return, in EAX, a value between zero and six denoting the day of the week of the given Gregorian date (0=sun, 1=mon, etc.)

HLA high-level calling sequence examples:

```
date.dayOfWeek( someDateVar );
```

```

mov( eax, dayOfWeek1 );
date.dayOfWeek( month, day, year );
mov( eax, dayOfWeek2 );

```

HLA low-level calling sequence examples:

```

push( someDateVar.date );
call date._dayOfWeek;
mov( eax, dayOfWeek1 );

```

## 11.6 Date Arithmetic

The functions in this category perform date arithmetic – adding integer (day) values to dates, subtracting integer (day) values, adding integer month or year values to a date, and computing the number of days between two dates.

```

#macro date.daysBetween
(
    m1:byte;
    d1:byte;
    y1:word;
    m2:byte;
    d2:byte;
    y2:word
); @returns( "eax" );

```

```

#macro date.daysBetween
(
    m1:byte;
    d1:byte;
    y1:word;
    dr:date.daterec
); @returns( "eax" );

```

```

#macro date.daysBetween
(
    dr:date.daterec;
    m:byte;
    d:byte;
    y:word
); @returns( "eax" );

```

```

#macro date.daysBetween
(
    dr1:date.daterec;
    dr2:date.daterec
); @returns( "eax" );

```

```

date._daysBetween( first:daterec; last:daterec ); @returns( "eax" );

```

These functions return an uns32 value in EAX that gives the number of days between the two specified dates. These functions work directly on the Gregorian dates.

HLA high-level calling sequence examples:

```
date.daysBetween( dateVar1, dateVar2 );
mov( eax, daysBetweenD1D2 );

date.daysBetween( m1, d1, y1, m2, d2, y2 );
mov( eax, daysBetween2 );

date.daysBetween( dateVar3, m4, d4, y4 );
mov( eax, daysBetween3 );

date.daysBetween( m5, d5, y5, dateVar6, );
mov( eax, daysBetween4 );
```

HLA low-level calling sequence examples:

```
push( dateVar1.date );
push( dateVar2.date );
call date._daysBetween;
mov( eax, daysBetween5 );
```

**date.addDays( days:uns32; var dr:date.daterec );**

This procedure adds the first parameter (in days) directly to the Gregorian date variable passed by reference as the second parameter.

HLA high-level calling sequence example:

```
date.addDays( days, someDateVar );
```

HLA low-level calling sequence example:

```
push( days );
push( someDateVar.date );
call date.addDays;
```

**date.addMonths( months:uns32; var dr:date.daterec );**

This procedure adds the first parameter (in months) directly to the Gregorian date variable passed by reference as the second parameter.

HLA high-level calling sequence example:

```
date.addMonths( months, someDateVar );
```

HLA low-level calling sequence example:

```
push( months);
push( someDateVar.date );
call date.addMonths;
```

**date.addYears( years:uns32; var dr:date.daterec );**

This procedure adds the first parameter (in years) directly to the Gregorian date variable passed by reference as the second parameter.

HLA high-level calling sequence example:

```
date.addYears( years, someDateVar );
```

HLA low-level calling sequence example:

```
push( years);
push( someDateVar.date );
call date.addYears;
```

**date.subDays( days:uns32; var dr:date.daterec );**

This procedure subtracts the first parameter (in days) directly from the Gregorian date variable passed by reference as the second parameter.

HLA high-level calling sequence example:

```
date.subDays( days, someDateVar );
```

HLA low-level calling sequence example:

```
push( days );
push( someDateVar.date );
call date.subDays;
```

**date.subMonths( months:uns32; var dr:date.daterec );**

This procedure subtracts the first parameter (in months) directly from the Gregorian date variable passed by reference as the second parameter.

HLA high-level calling sequence example:

```
date.subMonths( months, someDateVar );
```

HLA low-level calling sequence example:

```
push( months);
push( someDateVar.date );
call date.subMonths;
```

**date.subYears( years:uns32; var dr:date.daterec );**

This procedure subtracts the first parameter (in years) directly from the Gregorian date variable passed by reference as the second parameter.

HLA high-level calling sequence example:

```
date.subYears( years, someDateVar );
```

HLA low-level calling sequence example:

```
push( years);
push( someDateVar.date );
call date.subYears;
```

## 11.7 Reading the Current System Date

The functions in this category read the current date from the system.

**date.today( var dr:date.daterec );**

Stores the local date (today's date) into the specified parameter. Warning: some systems may not provide a localized date and time, if this is the case then this function will return the UTC/GMT date. If this would cause your application to fail, then you should read both the local and UTC dates and times and, if they are not different, apply an application-defined time zone difference to the local date value.

HLA high-level calling sequence example:

```
date.today( someDateVar );
```

HLA low-level calling sequence example:

```
// Assume that "someDateVar" is a static object:

pushd( &someDateVar );
call date.today;

// If someDateVar is not a static object:

lea( eax, someDateVar );
push( eax );
call date.today;
```

**date.utc( var dr:date.daterec );**

Stores the UTC date (today's GMT date) into the specified parameter. Of course, the difference between the local and GMT date depend entirely upon which time zone you're in.

HLA high-level calling sequence example:

```
date.utc( someDateVar );
```

HLA low-level calling sequence example:

```
// Assume that "someDateVar" is a static object:

pushd( &someDateVar );
call date.utc;

// If someDateVar is not a static object:

lea( eax, someDateVar );
push( eax );
call date.utc;
```

## 11.8 Date Output and String Conversion

The date module contains several functions that let you choose a date output format, convert a date to a string, and output dates (to the standard output device). This section describes those functions.

**date.setFormat( fmt : OutputFormat );**

This sets the internal format variable to the *date.OutputFormat* value you specify. This constant must be one of the *date.OutputFormat* enumerated constants (given earlier) or *date.SetFormat* will raise an *ex.InvalidDateFormat* exception.

HLA high-level calling sequence example:

```
date.setFormat( date.mmdyyy );
date.setFormat( dateFmtVariable );
```

HLA low-level calling sequence example:

```
// If the parameter is a constant:

pushd( date.mmdyyy );
call date.setFormat;

// If someFmtVar is byte variable and the
// three bytes following it are in paged memory:

push( (type dword someFmtVar) );
call date.setFormat;

// If you cannot access the three bytes beyond someFmtVar:

movzx( someFmtVar, eax );
push( eax );
call date.setFormat;
```

**date.setSeparator( chr:char );**

This procedure sets the internal date separator character (default is '/') to the character you pass as a parameter. This is used when printing dates and converting dates to strings.

HLA high-level calling sequence example:

```
date.setSeparator( '-' );
date.setSeparator( someCharVar );
```

HLA low-level calling sequence example:

```
// If the parameter is a constant:

pushd( '/' );
call date.setSeparator;

// If someFmtVar is byte variable and the
// three bytes following it are in paged memory:

push( (type dword someCharVar) );
call date.setSeparator;

// If you cannot access the three bytes beyond someFmtVar:
```

```

movzx( someCharVar, eax );
push( eax );
call date.setSeparator;

#macro date.toString( m:byte; d:byte; y:word; s:string );
#macro date.toString( dr:date.daterec; s:string);
date._toString( dr:daterec; s:string );

```

These functions will convert the specified date to a string (using the output format specified by *date.SetFormat* and the separator character specified by *date.SetSeparator*) and store the result in the specified string. An *ex.StringOverflow* exception occurs if the destination string's *MaxStrLen* field is too small (generally, 20 characters handles all string formats).

HLA high-level calling sequence examples:

```

date.toString( someDateVar, dateString1 );
date.toString( month, day, year, dateString2 );

```

HLA low-level calling sequence examples:

```

push( someDateVar.date );
push( dateString3 );
call date._toString;

```

```

#macro date.a_toString( m:byte; d:byte; y:word ); @returns( "eax" );
#macro date.a_toString( dr:date.daterec ); @returns( "eax" );
date._a_toString( dr:daterec ); @returns( "eax" );

```

These procedures are similar to the *date.toString* procedures above except they automatically allocate the storage for the string and return a pointer to the string object in the EAX register. You should free the string storage with *str.free* with you are done with this string.

HLA high-level calling sequence examples:

```

date.a_toString( someDateVar );
mov( eax, dateStr1 );
date.a_toString( month, day, year );
mov( eax, dateStr2 );

```

HLA low-level calling sequence examples:

```

push( someDateVar.date );
call date._a_toString;
mov( eax, dateStr3 );

```

## 11.9 Date Class Types

For those who prefer an object-oriented programming approach, the Standard Library provides the ability to create date class data types. To use one of the date class data types, you must include the following statement at the beginning of your HLA program:

```
#include( "dtClass.hhf" )
```

Note that *stdlib.hhf* does not include *dtClass.hhf*, so you must explicitly include the *dtClass.hhf* header file if you intend to use any of the date class functions or data types.

The Standard Library provides two predefined date class types: *dateClass\_t* and *virtualDateClass*. The difference between these two types is that the *dateClass\_t* type uses static procedures for all the date functions whereas *virtualDateClass\_t* uses virtual methods for all the date functions. In certain cases, using the *dateClass\_t* data type is more efficient than using *virtualDateClass\_t* because you only link in the class functions you actually call. However, you lose the object-oriented method inheritance/override ability when using the *dateClass\_t* type rather than the *virtualDateClass\_t* type. For more details on the differences between these two class types, please see the discussion of the *dtClass.make\_dateClass* macro appearing later in this section. This section will use the phrase "date class" to mean any class created by the *make.dateClass\_t* macro, including the *dateClass\_t* and *virtualDateClass\_t* data types.

The date class types provide three data fields:

```
var
    theDate:      date.daterec;
    OutFmt:       date.OutputFormat;
    Separator:    char;
```

The first field, *theDate*, holds the date value associated with the date object. This is the standard *date.daterec* date type described earlier in this document. Note that you can pass this field to any of the standard date and time functions that expect a *date.daterec* value.

The second field, *OutFmt*, specifies the output format when using the date class string conversion routines. Note that only the date class string conversion routines respect the value of this field; if you pass *theDate* directly to a date function that takes a *date.daterec* argument, that function will use the system-wide global date format rather than the object's *OutFmt* value.

**Thread Safety Issue:** Although each date object has its own *OutFmt* field, this does not make the use of date class objects thread safe. When converting the *theDate* to a string, the date class functions save the global format value, copy *OutFmt* to the global format value, call the date functions to do the string conversion, and then restore the original global value. If a thread is suspended during this activity then any date/string conversions during this suspension may use an incorrect format value. This issue will be corrected in a later version of the Standard Library. For now, you must manually protect all date/string conversions if you perform such conversions in multiple threads in your application.

The third field, *Separator*, holds the character that is used to separate the months, days, and years fields during a string conversion. The *dateClass\_t* and *virtualDateClass\_t* constructors initialize this field with a slash character ('/').

Of course, you may create a derived class from either *dateClass\_t* or *virtualDateClass\_t* (or create a brand new date class using the *dtClass.make\_dateClass* macro) and add any other fields you like to that new date class. One suggestion for such a class is to pad the data fields to a multiple of four bytes. Currently, the *dateClass\_t* and *virtualDateClass\_t* objects consumes ten bytes of storage (six bytes for the three fields above plus four bytes for the VMT pointer). For performance reasons, you might want to extend the size of the data storage to 12 or even 16 bytes.

## 11.9.1 Date Class Methods/Procedures

In most HLA classes, there are two types of functions: (static) procedures and (dynamic) methods (there are also iterators, but the date classes do not use iterators so we will ignore that here). The only difference between a method and a procedure is how the program actually calls the function: the program calls procedures directly, it calls methods indirectly through an entry in the virtual method table (VMT). Static procedure calls are very efficient, but you lose the benefits of inheritance and functional polymorphism when you define a function as a static procedure in a class. Methods, on the other hand, fully support polymorphic calls, but introduce some efficiency issues.

First of all, unlike static procedures, your program will link in all methods defined in your program *even if you don't explicitly call those methods in your program*. Because the call is indirect, there really is no way for the assembler and linker to determine whether you've actually called the function, so it must assume that you do call it and links in the code for each method in the class. This can make your program a little larger because it may be including several date class functions that you don't actually call.

The second efficiency issue concerning method calls is that they use the EDI register to make the indirect call (static procedure calls do not disturb the value in EDI). Therefore, you must ensure that EDI is free and available before making a virtual method call, or take the effort to preserve EDI's value across such a call.

A third, though extremely minor, efficiency issue concerning methods is that the class' VMT will need an extra entry in the virtual method table. As this is only four bytes per class (not per object), this probably isn't much of a concern.

The predefined *dateClass\_t* and *virtualDateClass\_t* classes differ in how they define the functions appearing in the class types. The *dateClass\_t* type uses static procedures for all functions, the *virtualDateClass\_t* type uses methods for all class functions. Therefore, *dateClass\_t* date types will make direct calls to all the functions (and



only link in the procedures you actually call); however, *dateClass\_t* objects do not support function polymorphism in derived classes. The *virtualDateClass\_t* type does support polymorphism for all the class methods, but whenever you use this data type you will link in all the methods (even if you don't call them all) and calls to these methods will require the use of the EDI register.

It is important to understand that *dateClass\_t* and *virtualDateClass\_t* are two separate types. Neither is derived from the other. Nor are the two types compatible with one another. You should take care not to confuse objects of these two types if you're using both types in the same program.

## 11.9.2 Creating New Date Class Types

As it turns out, the only difference between a method and a procedure (in HLA) is how that method/procedure is called. The actual function code is identical regardless of the declaration (the reason HLA supports method and procedure declarations is so that it can determine how to populate the VMT and to determine how to call the function). By pulling some tricks, it's quite possible to call a procedure using the method invocation scheme or call a method using a direct call (like a static procedure). The Standard Library date class module takes advantage of this trick to make it possible to create new date classes with a user-selectable set of procedures and methods. This allows you to create a custom date type that uses methods for those functions you want to override (as methods) and use procedures for those functions you don't call or will never override (as virtual methods). Indeed, the *dateClass\_t* and *virtualDateClass\_t* date types were created using this technique. The *dateClass\_t* data type was created specifying all functions as procedures, the *virtualDateClass\_t* data type was created specifying all functions as methods. By using the *dtClass.make\_dateClass* macro, you can create new date data types that have any combination of procedures and methods.

```
dtClass.make_dateClass( className, "<list of methods>" )
```

*dtClass.make\_dateClass* is a macro that generates a new data type. As such, you should only invoke this macro in an HLA type declaration section. This macro requires two arguments: a class name and a string containing the list of methods to use in the new data type. The method list string must contain a sequence of method names (typically separated by spaces, though this isn't strictly necessary) from the following list:

```
today
utc
isLeapYear
isValid

validate

a_toString
toString
setSeparator
setFormat
addDays
subDays
addMonths

subMonths

addYears

subYears
fromJulian
toJulian
dayOfWeek
dayNumber
daysLeft
daysBetween

difference
```

Here is *dtClass.make\_dateClass* macro invocation that creates the *virtualDateClass\_t* type:

```
type
    dtClass.make_dateClass
```

```
(
    virtualDateClass,
    "today"
    "isLeapYear"
    "isValid"
    "a_toString"
    "toString"
    "setSeparator"
    "setFormat"
    "addDays"
    "subDays"
    "addMonths"
    "addYears"
    "fromJulian"
    "toJulian"
    "dayOfWeek"
    "dayNumber"
    "daysLeft"
    "daysBetween"
);
```

(For those unfamiliar with the syntax, HLA automatically concatenates string literals that are separated by nothing but whitespace; therefore, this macro contains exactly two arguments, the `virtualDateClass_t` name and a single string containing the concatenation of all the strings above.)

From this macro invocation, HLA creates a new data type using methods for each of the names appearing in the string argument. If a particular date function's name is not present in the `dtClass.make_dateClass` macro invocation, then HLA creates a static procedure for that function. As a second example, consider the declaration of the `dateClass_t` data type (which uses static procedures for all the date functions):

```
type
    dtClass.make_dateClass( dateClass_t, " " );
```

Because the function string does not contain any of the date function names, the `dtClass.make_dateClass` macro generates static procedures for all the date functions.

The `dateClass_t` type is great if you don't need to create a derived date class that allows you to polymorphically override any of the date functions. If you do need to create methods for certain functions and you don't mind linking in all the date class functions (and you don't mind the extra overhead of a method call, even for those functions you're not overloading), the `virtualDateClass_t` is convenient to use because it makes all the functions virtual (that is, methods). Probably 99% of the time you won't be calling the date functions very often, so the overhead of using method invocations for all date functions is irrelevant. In those rare cases where you do need to support polymorphism for a few date functions but don't want to link in the entire set of date functions, or you don't want to pay the overhead for indirect calls to functions that are never polymorphic, you can create a new date class type that specifies exactly which functions require polymorphism.

For example, if you want to create a date class that overrides the definition of the `fromJulian` and `toJulian` functions, you could declare that new type thusly:

```
type
    dtClass.make_dateClass
    (
        myDateClass,
        "fromJulian"
        "toJulian"
    );
```

This new class type (`myDateClass`) has two methods, `fromJulian` and `toJulian`, and all the other date functions are static procedures. This allows you to create a derived class that overloads the `fromJulian` and `toJulian` methods and access those methods when using a generic `myDateClass` pointer, e.g.,

```
type
    derivedMyDateClass :
```

```

class inherits( myDateClass );

    override method fromJulian;
    override method toJulian;

endclass;

```

It is important for you to understand that types created by *dtClass.make\_dateClass* are base types. They are not derived from any other class (e.g., *virtualDateClass\_t* is not derived from *dateClass\_t* or vice-versa). The types created by the *dtClass.make\_dateClass* macro are independent and incompatible types. For this reason, you should avoid using different base date class types in your program. Pick (or create) a base date class and use that one exclusively in an application. You'll avoid confusion by following this rule.

For the sake of completeness, here are the macros that the Standard Library uses to create date data types:

```

namespace dtClass;

// The following macro allows us to turn a class function
// into either a method or a procedure based on the
// presence of "funcName" within a list of method names
// passed to the class generating macro.

#macro function( funcName );

    #if( @index( methods, 0, @string:funcName) = -1 )

        procedure funcName

    #else

        method funcName

    #endif

#endmacro

#macro make_dateClass( className, methods );

    className:
    class

        var
            theDate:    date.daterec;
            OutFmt:     date.OutputFormat;
            Separator:  char;

        procedure create;
            @external( "DATECLASS_CREATE" );

        dtClass.function( today );
            @external( "DATECLASS_TODAY" );

        dtClass.function( utc );
            @external( "DATECLASS_UTC" );

        dtClass.function( isLeapYear );
            @returns( "al" );
            @external( "DATECLASS_ISLEAPYEAR" );

        dtClass.function( isValid );

```

```

        @returns( "al" );
        @external( "DATECLASS_ISVALID" );

dtClass.function( validate );
    @returns( "al" );
    @external( "DATECLASS_VALIDATE" );

dtClass.function( a_toString );
    @returns( "eax" );
    @external( "DATECLASS_A_TOSTRING" );

dtClass.function( toString )( dest:string );
    @external( "DATECLASS_TOSTRING" );

dtClass.function( setSeparator )( c:char );
    @external( "DATECLASS_SETSEPARATOR" );

dtClass.function( setFormat )( f:date.OutputFormat );
    @external( "DATECLASS_SETFORMAT" );

dtClass.function( addDays )( days:uns32 );
    @external( "DATECLASS_ADDDDAYS" );

dtClass.function( subDays )( days:uns32 );
    @external( "DATECLASS_SUBDAYS" );

dtClass.function( addMonths )( months:uns32 );
    @external( "DATECLASS_ADDMONTHS" );

dtClass.function( subMonths )( days:uns32 );
    @external( "DATECLASS_SUBMONTHS" );

dtClass.function( addYears )( years:uns32 );
    @external( "DATECLASS_ADDYEARS" );

dtClass.function( subYears )( days:uns32 );
    @external( "DATECLASS_SUBYEARS" );

dtClass.function( fromJulian )( Julian:uns32 );
    @external( "DATECLASS_FROMJULIAN" );

dtClass.function( toJulian );
    @returns( "eax" );
    @external( "DATECLASS_TOJULIAN" );

dtClass.function( dayOfWeek );
    @returns( "eax" );
    @external( "DATECLASS_DAYOFWEEK" );

dtClass.function( dayNumber );
    @returns( "eax" );
    @external( "DATECLASS_DAYNUMBER" );

dtClass.function( daysLeft );
    @returns( "eax" );
    @external( "DATECLASS_DAYSLEFT" );

dtClass.function( daysBetween )
(
    otherDate:date.daterec
);

```

```

        @returns( "eax" );
        @external( "DATECLASS_DAYSBEETWEEN" );

        dtClass.function( difference )
        (
            var otherDate:className in eax
        );
        @returns( "eax" );
        @external( "DATECLASS_DIFFERENCE" );

        endclass
    #endmacro

end dtClass;

```

If you look closely at the *make\_dateClass* macro, you'll notice that it maps all the functions, be they methods or procedures, to the *dateClass\_t* names (which are all procedures, if you look at the source code for these functions). As noted earlier, the function code for methods and procedures is exactly the same, only the call to a given function is different based on whether it is a method or a procedure. Therefore, the *dtClass.make\_dateClass* macro maps all functions to the same set of procedures. Therefore, if you do create and use multiple date classes in the same application, the linker will only link in one set of routines (unless, of course, you overload some methods, in which case the linker will link in your new functions as well as the original *dateClass\_t* set).

### 11.9.3 Date Class Functions

The date class type supports most of the functions associated with the date type. The main difference is that the date class functions operate directly on the date object rather than on a date value you pass as a parameter. For this reason, there aren't any macros that overload the date function parameter lists.

The following sections do not include sample code demonstrating the calling sequences for a couple of reasons:

For high level calls, the syntax depends on the object name and type.

Low-level calling sequences don't appear here because it doesn't really make sense to make a low-level object invocation; people wanting to make low-level calls will probably use the standard date procedures rather than the object-oriented ones.

These functions are really intended for use by programmers experienced with HLA's Object-oriented assembly facilities. Note that the *dtClass.hhf* header file is not automatically included by *stdlib.hhf*; this reflects the more advanced nature of the date class module.

For the same reasons, there are no stack diagrams for these function calls. If you want more information on making calls to HLA class methods and procedures, please consult the HLA documentation.

In the following function descriptions, the symbol <object> is used to specify a date class object or a pointer to a date class object. Note that class invocations of static procedures (e.g., "dateClass\_t.isLeapYear") are illegal with the single exception of the constructor (the *create* procedure). If you call a date class procedure directly, the system will raise an exception (as ESI, which should be pointing at the object's data, will contain NULL).

#### <object>.create();

The <name>.create procedure is the object constructor. This is the only function that you may call using a class name rather than an object name. For example, *dateClass\_t.create()*; is a perfectly legitimate constructor call. As is the convention for HLA class constructors, if you call a class constructor directly (using the class name rather than an object name), the date class constructor will allocate storage for a new date class object on the heap and return a pointer to the new object in ESI. Once the storage is allocated (or if you specify the name of a previously-allocated object rather than the class name), the date class constructor will initialize all the fields of the object to reasonable values (in particular, the constructor initializes the VMT pointer, initializes *theDate* to a valid date, and sets up the *OutFmt* and *Separator* fields with default values).

If you create a derived date class and add new data fields to the data type, you should override the *create* procedure and initialize those new fields in the overridden procedure. See the HLA documentation or *The Art of Assembly Language* for more details on derived classes and overriding constructors.

```
<object>.isLeapYear(); @returns( "al" );
```

This function returns true or false in the AL register depending upon whether the object's *theDate* field is a leap year (true if it is a leap year). See the discussion of *date.isLeapYear* for more details.

```
<object>.validate();
```

This function checks the object's *theDate* field and raises an *ex.InvalidDate* exception if the date is not a valid date between 1/1/1600 and 12/31/2999. See the discussion of *date.validate* for more details.

```
<object>.isValid(); @returns( "al" );
```

This function checks the object's *theDate* field and returns false in EAX if the date is not a valid date between 1/1/1600 and 12/31/2999 (it returns true otherwise). See the discussion of *date.isValid* for more details.

```
<object>.toJulian(); @returns( "eax" );
```

This function converts the Gregorian (i.e., standard) date found in the object's *theDate* field into a Julian day number. See the discussion of *date.toJulian* for more details.

```
<object>.fromJulian( jd:uns32 );
```

This function converts the Julian Day Number passed as the argument to a Gregorian (i.e., standard) date and stores the result the object's *theDate* field. See the discussion of *date.fromJulian* for more details. This function will raise an *ex.InvalidDate* exception if the Julian Day Number conversion produces a date outside the range 1/1/1600 to 12/31/2999. See the discussion of *date.fromJulian* for more details.

```
<object>.dayNumber(); @returns( "eax" );
```

This function converts the Gregorian date found in the objects *theDate* field into a day number into the current year. It returns the day number in the EAX register. See the discussion of *date.dayNumber* for more details.

```
<object>.daysLeft(); @returns( "eax" );
```

This function returns the number of days left in the current year *counting the object's theDate field*. See the discussion of *date.daysLeft* for more details.

```
<object>.dayOfWeek(); @returns( "eax" );
```

These functions return, in EAX, a value between zero and six denoting the day of the week of *theDate*. See the discussion of *date.dayOfWeek* for more details.

```
<object>.daysBetween( otherDate:daterec ); @returns( "eax" );
```

This function returns an *uns32* value in EAX that gives the number of days between object's date and the *daterec* value passed as a parameter dates. See the discussion of *date.daysBetween* for more details.

```
<object>.difference( var otherDate:<classType> ); @returns( "eax" );
```

This function returns an *uns32* value in EAX that gives the number of days between object's date and the date class object value passed as a parameter. The parameter type ("*<classType>*") must be the same type as *<object>*. See the discussion of *date.daysBetween* for more details.

```
<object>.addDays( days:uns32 );
```

This procedure adds the parameter value (in days) directly to the object's *theDate* field. See the discussion of *date.addDays* for more details.

```
<object>.addMonths( months:uns32; var dr:date.daterec );
```

This procedure adds the parameter value (in months) directly to the object's *theDate* field. See the discussion of *date.addMonths* for more details.

```
<object>.addYears( years:uns32; var dr:date.daterec );
```

This procedure adds the parameter value (in years) directly to the object's *theDate* field. See the discussion of *date.addYears* for more details.

```
<object>.subDays( days:uns32; var dr:date.daterec );
```

This procedure subtracts the parameter value (in days) directly from the object's *theDate* field. See the discussion of *date.subDays* for more details.

```
<object>.subMonths( months:uns32; var dr:date.daterec );
```

This procedure subtracts the parameter value (in months) directly from the object's *theDate* field. See the discussion of *date.subMonths* for more details.

```
<object>.subYears( years:uns32; var dr:date.daterec );
```

This procedure subtracts the parameter value (in years) directly from the object's *theDate* field. See the discussion of *date.subYears* for more details.

```
<object>.today();
```

Stores the local date (today's date) into the object's *theDate* field. See the discussion of *date.today* for more details.

```
<object>.utc( var dr:date.daterec );
```

Stores the UTC date (today's GMT date) into the object's *theDate* field. See the discussion of *date.utc* for more details.

```
<object>.toString( s:string );
```

This function converts the object's *theDate* field to a string (using the output format specified by the object's *theDate* field and the separator character specified by the object's *OutFmt* field) and stores the result in the specified string. See the discussion of *date.toString* for more details.

```
<object>.a_toString( dr:daterec ); @returns( "eax" );
```

This function converts the object's *theDate* field to a string (using the output format specified by the object's *theDate* field and the separator character specified by the object's *OutFmt* field) and stores the result in storage it allocates on the heap. This function returns a pointer to the new string in the EAX register. See the discussion of *date.a\_toString* for more details.





## 12 Environment Variables Module (env.hhf)

The env module contains a couple of functions that fetch environment strings from the operating system. These functions can be used to test global values set by the user in the environment space inherited by your processes.

### 12.1 The Env Module

To use the environment functions in your application, you will need to include one of the following statements at the beginning of your HLA application:

```
#include( "env.hhf" )
or
#include( "stdlib.hhf" )
```

### 12.2 Retrieving Environment Strings

The env module contains two functions for retrieving environment string data. To each of these functions you must pass the name of an environment variable (in the envVar string parameter). These functions will locate the specified environment variable in the system (if it is present) and return the associated string value.

```
procedure env.get( envVar:string; dest:string );
```

*env.get* copies the environment string data to the (preallocated) string variable you specify via the dest parameter. Note that the dest parameter must contain the address of a value HLA string object or this function may fail (or raise an exception). If the allocated storage isn't large enough to hold the string data, or the environment variable's string is greater than 4095 characters long, this function will raise an exception. This function returns true in EAX if it locates the environment variable in the environment space and successfully returns the environment variable's data. This function returns false in EAX if it cannot locate the environment variable in the environment space.

HLA high-level calling sequence examples:

```
env.get( envVarName, envData );
if( eax ) then

    // process the environment data in envData

endif;
```

HLA low-level calling sequence examples:

```
push( envVarName );// Assumption: envVarName is a string variable
call env.get;
```

```
procedure env.a_get( envVar:string );
```

*env.a\_get* also returns the value of an environment variable; however, it allocates storage for the result and returns a pointer to the string result (on the heap) in the EAX register, if such an environment variable exists. This function returns NULL in EAX if the environment variable does not exist. This function raises an exception if the environment variable's data is greater than 4095 characters long. The caller is responsible for freeing the storage allocated on the heap by this function.

HLA high-level calling sequence examples:

```
env.a_get( envVarName );  
mov( eax, envData );  
if( eax <> NULL ) then  
  
    // process the environment data pointed at by envData  
  
endif;  
str.free( envData );
```

HLA low-level calling sequence examples:

```
push( envVarName );// Assumption: envVarName is a string variable  
call env.get;  
mov( eax, envData );
```

## 13 Exceptions Module (excepts.hhf)

The exceptions module contains several things of interest. First, it defines the `ExceptionValues` enumerated data type that lists out all the standard exceptions in the HLA Standard Library. The second thing provided in the `excepts` unit is the `ex.PrintExceptionError` procedure which prints a string associated with the exception number in `EAX`. Next, the `excepts.hhf` header file defines the `"assert( expr )"` macro. Finally, the `excepts.hhf` header file defines some procedures that the HLA run-time system uses to maintain the exception handling system; these procedures are of interest only to those who want to override the default HLA exception handling mechanisms.

### 13.1 The Exceptions Module

To use the exceptions functions in your application, you will need to include one of the following statements at the beginning of your HLA application:

```
#include( "excepts.hhf" )
or
#include( "stdlib.hhf" )
```

### 13.2 Exception Resource Reduction

By default, if you include `excepts.hhf` or `stdlib.hhf` in your HLA main program, HLA will automatically link in a set of strings that describe, in detail, each of the possible exceptions. This string data consumes a considerable amount of space and may not be necessary if you're not taking advantage of HLA's exception-handling system.

HLA will link in these strings if the compile-time variable `@exceptions` contains true when HLA encounters the **begin** associated with the main program. If you would like HLA to link in a single (small) string in place of this huge table of strings, just set the `@exceptions` compile-time variable to false after include `excepts.hhf` (or `stdlib.hhf`), e.g.,

```
#include( "excepts.hhf" )
?@exceptions := false;
```

This will reduce the size of your executable. Note, however, that you'll get a single "unhandled exception" error message if an unhandled error ever comes along. So during debugging, it's probably best to leave the exception strings in the program and remove them only for a release of your program.

### 13.3 Exception Constants

The following paragraphs describe each of the standard HLA exception constants and describe the conditions that lead to the Standard Library routines raising these exceptions. The `"excepts.hhf"` header file defines these constants. Since this list changes frequently, please refer to the `excepts.hhf` header file for the most recent list of exception names. HLA and the HLA Standard Library only raise these exceptions; user applications, however, may define other exceptions in addition to these. Of course, user applications may also raise exceptions using these exception constants. Note that the following numeric constants for the exception names are subject to change.

```
/* 0 */UnknownException,

// String related exceptions:

/* 1 */StringOverflow,
/* 2 */StringIndexError,
/* 3 */StringOverlap,
/* 4 */StringMetaData,
/* 5 */StringAlignment,
/* 6 */StringUnderflow,
/* 7 */IllegalStringOperation,

// General exceptions:
```

```

/* 8 */ValueOutOfRange,
/* 9 */IllegalChar,
/* 10 */AttemptToDerefNULL,
/* 11 */TooManyCmdLnParms,
/* 12 */AssertionFailed,
/* 13 */ExecutedAbstract,
/* 14 */BadObjPtr,
/* 15 */InvalidAlignment,
/* 16 */InvalidArgument,
/* 17 */BufferOverflow,
/* 18 */BufferUnderflow,
/* 19 */IllegalSize,

// Formatting and conversion errors:

/* 20 */ConversionError,
/* 21 */WidthTooBig,
/* 22 */FractionTooBig,

// File-related exceptions:

/* 23 */BadFileHandle,
/* 24 */FileNotFound,
/* 25 */FileOpenFailure,
/* 26 */FileCloseError,
/* 27 */FileWriteError,
/* 28 */FileReadError,
/* 29 */FileSeekError,
/* 30 */DiskFullError,
/* 31 */AccessDenied,
/* 32 */EndOfFile,

// FileSys-related exceptions:

/* 33 */CannotCreateDir,
/* 34 */CannotRemoveDir,
/* 35 */CannotRemoveFile,
/* 36 */CDFailed,
/* 37 */CannotRenameFile,

// Memory management exceptions:

/* 38 */MemoryAllocationFailure,
/* 39 */MemoryFreeFailure,
/* 40 */MemoryAllocationCorruption,
/* 41 */AttemptToFreeNULL,
/* 42 */BlockAlreadyFree,
/* 43 */CannotFreeMemory,
/* 44 */PointerNotInHeap,

// Array exceptions:

/* 45 */ArrayShapeViolation,
/* 46 */ArrayBounds,

// Time/date exceptions:

/* 47 */InvalidDate,
/* 48 */InvalidDateFormat,
/* 49 */TimeOverflow,
/* 50 */InvalidTime,

```

```

/* 51 */InvalidTimeFormat,

// Socket Errors:

/* 52 */SocketError,

// Thread Errors:

/* 53 */ThreadError,

// Hardware/OS related exceptions

/* 54 */AccessViolation,
/* 55 */InPageError,
/* 56 */NoMemory,
/* 57 */InvalidHandle,
/* 58 */ControlC,
/* 59 */StackOverflow,
/* 60 */Breakpoint,
/* 61 */SingleStep,
/* 62 */PrivInstr,
/* 63 */IllegalInstr,
/* 64 */BoundInstr,
/* 65 */IntoInstr,
/* 66 */DivideError,
/* 67 */fDivByZero,
/* 68 */fInexactResult,
/* 69 */fInvalidOperation,
/* 70 */fOverflow,
/* 71 */fUnderflow,
/* 72 */fStackCheck,
/* 73 */fDenormal,

// Blob related exceptions

/* 74 */BlobOverflow

```

**ex.UnknownException**

This is a reserved value that HLA's Standard Library functions do not raise. The HLA run-time system displays this exception value if it cannot figure out the source of the interrupt. `ex.PrintExceptionError` calls also use this value to display an appropriate message for unhandled user exceptions.

**ex.StringOverflow**

The string functions in the HLA Standard Library raise this exception if the caller attempts to store too many characters into a string variable (causing a string overflow error).

**ex.StringIndexError**

Some string functions require a parameter that supplies an index into a string. If those functions require that the index be within the range `0..length-1`, they will raise this exception to denote an index out of range error.

**ex.ValueOutOfRange**

Several HLA Standard Library routines raise this exception if an integer calculation overflows. The best examples are the integer input routines (e.g., `stdin.geti8`) that will raise this exception if the user's input is otherwise legal but out of range for the specific data type (i.e., `-128..+127` for `stdin.geti8`).

**ex.IllegalChar**

Certain input and conversion routines raise this exception if an unexpected character comes along. An unexpected character is usually a non-ASCII character (character codes in the range \$80..\$FF). Note that the conversion and input routines do not raise this exception if a non-digit character comes along. See *ex.ConversionError* to see how the HLA Standard Library handles that exception.

**ex.AttemptToDerefNULL**

Many HLA Standard Library routines expect a pointer to some object as a parameter. If they do not allow a NULL pointer value (zero) the routines may explicitly test for a NULL value and raise this exception if the user inadvertently passes in a NULL pointer. Also see the *ex.AccessViolation* exception.

**ex.TooManyCmdLnParms**

The *args.hhf* module raises this exception if you specify too many command line parameters. The exact maximum value may vary between versions of the HLA Standard Library, but it's typically a value like 64 or 128.

**ex.AssertionFailed**

The HLA *assert* statement raises this expression if the value of the assertion expression evaluates false. See the section on assertions later in this section for more details.

**ex.ExecutedAbstract**

The HLA run-time system raises this exception if you attempt to execute an abstract class method that has not been overridden and defined.

**ex.BadObjPtr**

The HLA run-time system raises this exception if you attempt to execute a class method using an illegal pointer.

**ex.InvalidAlignment**

HLA raises this exception if you specify an illegal alignment value for an allocation operation.

**ex.ConversionError**

HLA raises this exception whenever there is some sort of error converting data from one from to another (usually, this exception occurs when converting string data to numeric data). For example, when converting a string to an integer value, the HLA Standard Library will raise this exception if it encounters a character that is not legal for that numeric type and is not a delimiter character.

**ex.WidthTooBig**

Certain numeric conversion and output functions let you specify a field width value for the conversion. Those routines raise this exception if that field width value is too large (this is nominally 256, but the exact value may be different).

**ex.BadFileHandle**

The file class and fileio library modules raise this exception if you attempt to read from or write to a file with an illegal file handle (i.e., the file has not been opened or has already been closed).

**ex.FileNotFound**

HLA raises this exception if you attempt to open (or otherwise access by name) a file and the system could not find the path/filename you specified.

**ex.FileOpenFailure**

The HLA file open routines raise this error if there was a catastrophic error opening a file.

**ex.FileCloseError**

The HLA file close routines raise this error if there was an error closing a file.

**ex.FileWriteError**

The HLA Standard Library file output routines raise this exception if there is an error while attempting to write data to a file. This is usually a catastrophic error such as file I/O or some hardware error.

**ex.FileReadError**

The HLA Standard Library file output routines raise this exception if there is an error while attempting to read data from a file. This is usually a catastrophic error such as file I/O or some hardware error.

**ex.FileSeekError**

The HLA Standard Library file routines raise this exception if there was an error while attempting to seek to some new position in a file.

**ex.DiskFullError**

The HLA Standard Library raises this exception if you attempt to write data to a disk that is full.

**ex.AccessDenied**

The HLA Standard Library raises this exception if you attempt to access a file for which you do not have proper access permission.

**ex.EndOfFile**

The HLA Standard Library file I/O routines raise this exception if you attempt to read data from a file after you've reached the end of file. Note that HLA does not raise this exception upon reaching the EOF. You must actually attempt to read beyond the end of the file.

**ex.CannotCreateDir**

The HLA Standard Library mkdir function raises this exception if you attempt to create a subdirectory and the system returns an error.

**ex.CannotRemoveDir**

The HLA Standard Library rmdir function raises this exception if you attempt to remove a subdirectory and the system returns an error.

**ex.CannotRemoveFile**

The HLA Standard Library rmdir function raises this exception if you attempt to remove a file and the system returns an error.

**ex.CDFailed**

The HLA Standard Library raises this exception if you attempt to switch to a new working directory and the system could not find that directory or the change working directory operation otherwise failed.

**ex.CannotRenameFile**

The HLA Standard Library raises this exception if you attempt to rename a file and the operation failed.

**ex.MemoryAllocationFailure**

HLA raises this exception if a function attempts to allocate storage and the memory allocation operation fails (because of insufficient storage).

**ex.MemoryFreeFailure**

HLA raises this exception if you attempt to free storage and the request could not be satisfied (see also: `CannotFreeMemory`).

**ex.MemoryFreeFailure**

HLA raises this exception if you attempt to free storage and the request could not be satisfied (see also: `CannotFreeMemory`).

**ex.AttemptToFreeNULL**

HLA raises this exception if you attempt to free storage storage but you pass a NULL pointer to be freed.

**ex.BlockAlreadyFree**

HLA raises this exception if you attempt to free storage storage that has already been freed.

**ex.CannotFreeMemory**

The HLA memory free routines raise this exception if there is an error deallocating memory that was (presumably) allocated earlier.

**ex.PointerNotInHeap**

The HLA memory management routines raise this exception if you pass a pointer to an object that is supposed to be on the heap, but the pointer does not reference any object on the heap.

**ex.ArrayShapeViolation**

The `arrays.hhf` module raise this exception if you attempt to copy data from one array to another or otherwise operate on two arrays with incompatible "shapes." The "shape" of an array is the number of dimensions and the bounds on each dimension of that array. Compatible arrays typically have the same number of dimensions and the same bounds on each dimensions (though there are some exceptions to this rule).

**ex.ArrayBounds**

The `arrays.hhf` module raises this exception if you attempt to supply the wrong number of array dimensions or one of the array indices is out of bounds for that array.

**ex.InvalidDate**

The HLA `datetime.hhf` module raises this expression if you supply an illegal date to a date function. Note that legal dates must fall between Jan 1, 1600 and Dec 31, 9999 and must have valid day and month values (depending on the month and year).

**ex.InvalidDateFormat**

The HLA date conversion routines raise this exception if the internal date format value is illegal.



**ex.TimeOverflow**

The HLA datetime.hhf module raises this exception if, during a time calculation, an overflow occurs.

**ex.InvalidTime**

The HLA datetime.hhf module raises this expression if you supply an illegal time to a time function. Note that legal times must fall between 00:00:00 and 23:59:59.

**ex.InvalidTimeFormat**

The HLA time conversion routines raise this exception if the internal time format value is illegal.

**ex.AccessViolation**

This is a hardware exception that the CPU raises if you attempt to access an illegal memory or I/O location.

**ex.InPageError**

This is a hardware exception that the CPU raises if you attempt to access an illegal memory or I/O location.

**ex.NoMemory**

This is an exception that the OS raises if it cannot provide memory for the requested operation.

**ex.InvalidHandle**

This is an exception that the OS raises if it you pass it an invalid handle value for some operation.

**ex.ControlC**

If control-C checking is enabled, Windows will raise this exception whenever the user presses control-C on the console device.

**ex.StackOverflow**

The OS raises this exception if the hardware (80x86) stack exceeds the bounds set by the linker.

**ex.Breakpoint**

This is a hardware exception that the CPU raises if you execute an INT 3 (breakpoint) instruction.

**ex.SingleStep**

This is a hardware exception that the CPU raises after each instruction if the trace flag is set in the EFLAGS register.

**ex.PrivInstr**

This is a hardware exception that the CPU raises if you attempt to execute a privileged instruction while in user (non-kernel) mode.

**ex.IllegalInstr**

This is a hardware exception that the CPU raises if you attempt to execute an opcode that is not a legal 80x86 instruction.

**ex.BoundInstr**

This is a hardware exception that the CPU raises if you execute a BOUND instruction and the register value is not within the bounds specified by the BOUND memory operand(s).

**ex.IntoInstr**

This is a hardware exception that the CPU raises if you execute an INTO instruction and the overflow flag is set.

**ex.DivideError**

This is a hardware exception that the CPU raises if you attempt to divide by zero or if the quotient will not fit in the destination operand.

**ex.fDivByZero**

This is a hardware exception that the FPU raises if you've enable floating point exceptions and a floating point division by zero occurs.

**ex.fInexactResult**

This is a hardware exception that the FPU raises if you've enable floating point exceptions and a floating point operation produces an inexact result.

**ex.fInvalidOperation**

This is a hardware exception that the FPU raises if you've enable floating point exceptions and you attempt an illegal operation on the FPU.

**ex.fOverflow**

This is a hardware exception that the FPU raises if you've enable floating point exceptions and a floating point operation produces an overflow (see *ex.fDenormal* and *ex.fUnderflow* for underflows).

**ex.fUnderflow**

This is a hardware exception that the FPU raises if you've enable floating point exceptions and an underflow occurs.

**ex.fStackCheck**

This is a hardware exception that the FPU raises if you've enable floating point exceptions and an FPU stack overflow occurs.

**ex.fDenormal**

This is a hardware exception that the FPU raises if you've enable floating point exceptions and a floating point operation produces a demormalized result.

## 13.4 Exception Messages

The exceptions module provides two functions for converting exception numbers into meaningful messages.

```
procedure ex.exceptionMsg( exceptionCode:dword; msg:string );
```

*ex.exceptionMsg* converts the exception code passed in the *exceptionCode* parameter to a string message and stores the resulting string into the string pointed at by the *msg* parameter. The *msg* string **must** be large enough to hold the result (128 characters should be sufficient). Note that this function cannot raise any exceptions because it may be called from inside an exception handler, hence the requirement that *msg* be of sufficient size to hold the string.

Note: no "ex.a\_exceptionMsg" function exists because the error code resulting in the call to the function might be an "out of memory" error and it wouldn't do to have this function produce an error.

If the exception code is outside the range of the valid exception codes, this function returns the message associated with the "unknown exception" code.

HLA high-level calling sequence examples:

```
static
  msg:str.strvar(256);
  .
  .
  .
  ex.exceptionMsg( someCode, msg );
  stdout.put( msg );
```

```
procedure ex.printStackTrace;
```

*ex.printStackTrace* displays the message associated with the exception code in EAX in a form appropriate to the OS (e.g., under Windows this brings up a dialog box, under Linux this prints the message to the standard error device).



## 14 File Class Module (fileclass.hhf)

The HLA Standard Library provides an object-oriented file access mechanism implemented via the `file_t` and `virtualfile_t` classes. Unless otherwise specified, this document will use the term "file class" to describe the generic file class rather than the specific instance of the `file_t` class (which uses static linking for all functions).

**Note:** HLA also provides a `fileio` library module that does file I/O using traditional procedures rather than class objects. If you're more comfortable using such a programming paradigm, or you prefer your code to be a bit more efficient, you should use the `fileio` module.

**Note:** Currently, the `file_t` class is implemented as a thin layer over the `fileio` module. That is, functions in the file class simply pass their parameters on to the corresponding functions in the `fileio` module. The ultimate intent, however, is for the `file_t` class to implement buffered I/O to improve performance. Because of the wide variety of operating systems that the HLA Standard Library supports (and will support), this may lead to some functionality limitations in future versions of the `file_t` class. In particular, you should only use `file_t` class objects to access files on block structured (disk) devices and avoid accessing character-oriented or other device types. Also, `file_t` objects will provide the best performance for sequential files. Though the intent is to fully support random-access to file data via `file_t` objects, you may get better performance by using the traditional file I/O functions in the `fileio` module.

**Note:** the `virtualFile_t` class is completely different from the `file_t` class. In particular, it is not a thin layer over the `fileio` module. All of the functions in the `virtualFile_t` class ultimately call the `virtualFile_t.read` and `virtualFile_t.write` functions to do file I/O. If you override these two functions (`read` and `write`), you will override the behavior of all methods in the `virtualFile_t` class. Note that this is not true for `file_t` objects.

**Warning:** *Don't forget that HLA objects modify the values in the ESI and EDI registers whenever you call a class procedure, method, or iterator. Do not leave any important values in either of these register when making calls to the following routines. If the use of ESI and EDI is a problem for you, you might consider using the fileio module that does not suffer from this problem.*

**A Note About Thread Safety:** The `file` class functions and the operating system maintain system-wide values to track things like file position within a file. Currently, these values apply to all threads in a process (and, in the case of the OS, all processes in the system). When accessing the same `file` object from different threads, you should use synchronization to serialize access to the `file` object.

**Note about function overloading:** the functions in the `file` classes use function overloading in order to allow you to specify the parameter lists in different ways. The macro that handles the overloading generally coerces the possible parameter types into a single object that it passes to the underlying function. The documentation for the specific functions will tell you whether a symbol is a macro or a function. For the most part, this should matter to you unless you are taking the address of a function (which you cannot do with a macro). See the HLA documentation for more details on function overloading via macros.

### 14.1 File Class Methods/Procedures

In most HLA classes, there are three types of functions: (static) procedures, (dynamic) methods, and dynamic iterators. The only difference between a method and a procedure is how the program actually calls the function: the program calls procedures directly, it calls methods indirectly through an entry in the virtual method table (VMT). The system always calls iterators indirectly through the VMT, so we will not consider them further. Static procedure calls are very efficient, but you lose the benefits of inheritance and functional polymorphism when you define a function as a static procedure in a class. Methods, on the other hand, fully support polymorphic calls, but introduce some efficiency issues.

First of all, unlike static procedures, your program will link in all methods defined in your program *even if you don't explicitly call those methods in your program*. Because the call is indirect, there really is no way for the assembler and linker to determine whether you've actually called the function, so it must assume that you do call it and links in the code for each method in the class. This can make your program a little larger because it may be including several date class functions that you don't actually call.

The second efficiency issue concerning method calls is that they use the EDI register to make the indirect call (static procedure calls do not disturb the value in EDI). Therefore, you must ensure that EDI is free and available before making a virtual method call, or take the effort to preserve EDI's value across such a call.

A third, though extremely minor, efficiency issue concerning methods is that the class' VMT will need an extra entry in the virtual method table. As this is only four bytes per class (not per object), this probably isn't much of a concern.

The predefined `file_t` and `virtualFile_t` objects differ in how they define the functions appearing in the class types. The `file_t` type uses static procedures for all functions, the `virtualFile_t` type uses methods for all class functions. Therefore, `file_t` object types will make direct calls to all the functions (and only link in the procedures you actually call); however, `file_t` objects do not support function polymorphism in derived classes. The `virtualFile_t` type does support polymorphism for all the class methods, but whenever you use this data type

you will link in all the methods (even if you don't call them all) and calls to these methods will require the use of the EDI register.

It is important to understand that *file\_t* and *virtualFile\_t* are two separate types. Neither is derived from the other. Nor are the two types compatible with one another. You should take care not to confuse objects of these two types if you're using both types in the same program.

## 14.2 A Quick Note

The following sections do not include sample code demonstrating the calling sequences for a couple of reasons:

For high level calls, the syntax depends on the object name and type.

Low-level calling sequences don't appear here because it doesn't really make sense to make a low-level object invocation; people wanting to make low-level calls will probably use the standard

*fileio* procedures rather than the object-oriented ones.

These functions are really intended for use by programmers experienced with HLA's Object-oriented assembly facilities.

For the same reasons, there are no stack diagrams for these function calls. If you want more information on making calls to HLA class methods and procedures, please consult the HLA documentation.

In the following function descriptions, the symbol <object> is used to specify a file class object or a pointer to a *file\_t* class object. This wherever this document uses the name "file\_t", you may substitute (as appropriate) "virtualFile\_t". Note that class invocations of static procedures (e.g., "file\_t.open") are illegal with the single exception of the constructor (the create procedure). If you call a file class procedure directly, the system will raise an exception (as ESI, which should be pointing at the object's data, will contain NULL).

## 14.3 General File Operations

The functions in this category let you initialize file objects, access fields of the file objects, and perform other housekeeping tasks.

```
<object>.create; @returns( "esi" );
file_t.create; @returns( "esi" );           [to create dynamic objects]
virtualFile_t.create; @returns( "esi" );   [to create dynamic objects]
```

The file class provides a *file\_t.create* or *virtualFile\_t.create* constructor which you should always call before making use of a file variable. For file variables (as opposed to file pointer variables), you should call this routine specifying the name of the file variable. For file pointer variables, you should call this routine using the class name and store the pointer returned in EAX into your file variable. For example, to initialize the two following two file objects, you would use code like the following:

```
var
  MyOutputFile: file_t;
  filePtr:      pointer to file_t;
  .
  .
  .

  MyOutputFile.create();

  file_t.create();
  mov( eax, filePtr );
```

Note that the *file\_t.create* constructor simply initializes the virtual method table pointer and does other necessary internal initialization. The constructor does not open a file or perform other file-related activities.

```
<object>.handle; @returns( "eax" );
```

This function returns the file *handle* in the EAX register. The returned value is invalid if you have not opened the file. You can pass this handle value to any of the Standard Library file routines (e.g., *fileio.putc*) that expect a handle. You may also pass this value to OS API functions that expect a file handle.

HLA high-level calling sequence examples:

```
filePtr.handle();
mov( eax, fileHandle );
```

## 14.4 Opening and Closing Files

```
<object>.open( filename:string; access:dword )
```

This method opens an existing file. The *filename* parameter is a string specifying the name of the file you wish to open. The *access* parameter is one of the following:

- fileio.r
- fileio.w
- fileio.rw
- fileio.a

The *fileio.r* constant tells *<object>.open* to open the file for read-only access. The *fileio.w* constant tells *<object>.open* to open the file for writing. Using the *fileio.rw* constant tells *<object>.open* to open the file for reading and writing. The *fileio.a* option tells the *<object>.open* function to open the file for writing and append all written data to the end of the file.

Before accessing the data in a file, you must open the file (which initializes the file handle). The *<object>.open* and *<object>.openNew* methods are excellent tools for this purpose. You may also open the file using direct calls to the OS API, but you must initialize the *<object>.fileHandle* field of the class variable before making any other method calls in the file class.

HLA high-level calling sequence examples:

```
filePtr.open( "myfile.txt", fileio.r );

// Note: the Access parameter is almost always a constant in
// calls to fileio.open. However, if you want to pass a variable
// value or a register value in this parameter, you may certainly
// do so:

MyOutputFile.open( filenameStr, accessVarByte );

filePtr.open( someStr, al );
```

```
<object>.openNew( filename:string )
```

This function opens a new file for writing (if the file already exists, it is first deleted and then a new file is opened for writing). The file is given the "normal" attribute.

Before accessing the data in a file, you must open the file (which initializes the file handle). The *<object>.open* and *<object>.openNew* methods are excellent tools for this purpose. You may also open the file using direct calls to the OS API, but you must initialize the *<object>.fileHandle* field of the class variable before making any other method calls in the file class.

HLA high-level calling sequence examples:

```
filePtr.openNew( "myfile.txt" );

// If the filename string pointer is in a register (EAX):

MyOutputFile.openNew( eax );
```

#### **<object>.close;**

This method closes a file opened via *<object>.open* or *<object>.openNew* and flushes any buffered data to the disk.

HLA high-level calling sequence examples:

```
filePtr.close();
MyOutputFile.close();
```

## 14.5 File Predicates

The functions in this category test conditions associated with the file.

#### **<object>.eof(); @returns( "al" );**

This function returns true in the AL register if the file pointer is at the end of the file. It returns false if the program can read additional data from the file.

**Warning:** *<object>.eof* only functions properly for actual disk files. If you attempt to read data from an interactive device like the system console (keyboard) or a serial port, *<object>.eof*'s behavior is incorrect (it will wind up eating a character from the interactive input stream every time you call it). Unfortunately, none of the Oses that HLA supports provide a way to test for EOF until after you've actually read a character from the input stream. A better solution, which works fine with both interactive input streams and file data is to use HLA's *try..endtry* statement to trap and EOF error when it occurs. For example, rather than writing the following:

```
while( !filePtr.eof( someHandle )) do
.
.
.
endwhile;
```

You should write the following:

```
try
  forever
    .
    .
    .
  endfor;
  exception( ex.EndOfFile );

endtry;
```

Note: under Windows, *<object>.eof* always returns false for character device files (e.g., keyboard input) and it returns false for all other non-disk file device types. Note that if the user presses ctrl-Z on the keyboard, *<object>.eof* will not return true, but the system will return an *ex.endOfFile* exception. If there is any chance you'll be reading data from a device file rather than a disk file, always use the *try..endtry* block to test for EOF.

HLA high-level calling sequence examples:



```
while( !filePtr.eof( fileHandle ) ) do

    <<something while not at EOF>>

endwhile;
```

```
<object>.eoln(); @returns( "al" );
```

This function returns true in AL if the file pointer is currently pointing at the OS' end-of-line sequence in the file (carriage return/line feed for Windows, linefeed for other operating systems).

HLA high-level calling sequence examples:

```
filePtr.eoln();
```

## 14.6 Miscellaneous Output

The following file output routines all assume that you've opened the <object> file variable via a call to <object>.open and you've successfully opened the file for output.

```
<object>.write( var buffer:var; count:dword )
```

This method writes the number of bytes specified by the *count* parameter to the file. The bytes starting at the address of the *buffer* byte are written to the file. No range checking is done on the *buffer*, it is your responsibility to ensure that the buffer contains at least *count* valid data bytes.

Note: Notice that the *buffer* parameter is an untyped reference parameter. Untyped reference parameters have special properties, so be sure to read the chapter on "Passing Parameters to Standard Library Routines" (parmpassing.rtf) if you are not absolutely sure you understand how untyped reference parameters operate.

HLA high-level calling sequence examples:

```
filePtr.write( buffer, count );

// If bufPtr is a dword object containing the
// address of the buffer whose data you wish to
// write to the file:

filePtr.write( val bufPtr, count );

// The following writes the four-byte value of
// the bufPtr variable to the file (an unusual
// operation):

filePtr.write( bufPtr, 4 );
```

```
<object>.putbool( b:boolean );
```

This procedure writes the string "true" or "false" to the <object> output file depending on the value of the *b* parameter.

HLA high-level calling sequence examples:

```
filePtr.putbool( boolVar );

// If the boolean is in a register (AL):

MyOutputFile.putbool( al );
```

**<object>.newln( );**

This function writes a newline sequence (carriage return/line feed under Windows, linefeed under other operating systems) to the specified output file (<object>).

HLA high-level calling sequence examples:

```
filePtr.newln();
MyOutputFile.newln();
```

## 14.7 Character, Character Set, and String Output

The following file output routines all assume that you've opened the <object> file variable via a call to <object>.open and you've successfully opened the file for output.

**<object>.putc( c:char )**

Writes the character specified by the *c* parameter to the file.

HLA high-level calling sequence examples:

```
filePtr.putc( charVar );

// If the character is in a register (AL):

MyOutputFile.putc( al );
```

**<object>.putcSize( c:char; width:int32; fill:char )**

Outputs the character *c* to the file filevar using at least *width* output positions. If the absolute value of *width* is greater than one, then this function writes *fill* characters as padding characters during the output. If *width* is a positive value greater than one, then <object>.putcSize writes *c* left justified in a field of *width* characters; if *width* is a negative value less than one, then <object>.putcSize writes *c* right justified in a field of *width* characters.

HLA high-level calling sequence examples:

```
filePtr.putcSize( charVar, width, padChar );
```

**<object>.putcset( cst:cset );**

This function writes all the members of the *cst* character set parameter to the specified file variable.

HLA high-level calling sequence examples:

```
filePtr.putcset( csVar );
MyOutputFile.putcset( [ebx] ); // EBX points at the cset.
```

**<object>.puts( s:string );**

This procedure writes the value of the string parameter to the file.

HLA high-level calling sequence examples:

```
filePtr.puts( strVar );
filePtr.puts( ebx ); // EBX holds a string value.
MyOutputFile.puts( "Hello World" );
```

**<object>.putsSize( s:string; width:int32; fill:char )**

This function writes the *s* string to the file using at least *width* character positions. If the absolute value of *width* is less than or equal to the length of *s*, then this function behaves exactly like *<object>.puts*. On the other hand, if the absolute value of *width* is greater than the length of *s*, then *<object>.putsSize* writes *width* characters to the output file. This procedure emits the *fill* character in the extra print positions. If *width* is positive, then *<object>.putsSize* right justifies the string in the print field. If *width* is negative, then *<object>.putsSize* left justifies the string in the print field. Generally, people expect the string to be left justified, so you should ensure that this value is negative to achieve this.

HLA high-level calling sequence examples:

```
filePtr.putsSize( strVar, width, ' ' );

// For the following, EBX holds the string value,
// ECX contains the width, and AL holds the pad
// character:

MyOutputFile.putsSize( ebx, ecx, al );

filePtr.putsSize( "Hello World", 25, padChar );
```

## 14.8 Hexadecimal Numeric Output

The following file output routines all assume that you've opened the *<object>* file variable via a call to *<object>.open* and you've successfully opened the file for output.

**<object>.putb( b:byte );**

This procedure writes the value of *b* to the file using exactly two hexadecimal digits (including a leading zero if necessary).

HLA high-level calling sequence examples:

```
filePtr.putb( byteVar );

// If the character is in a register (AL):
```

```
MyOutputFile.putb( al );
```

**<object>.puth8( b:byte );**

This procedure writes the value of *b* to the file using one or two hexadecimal digits (the minimum necessary).

HLA high-level calling sequence examples:

```
filePtr.puth8( byteVar );

// If the character is in a register (AL):

MyOutputFile.puth8( al );
```

**<object>.puth8Size( b:byte; width:dword; fill:char )**

This procedure writes the value of *b* to the file using a minimum field *width* and a *fill* character. See the discussion of *width* and *fill* in the description of the *<object>.putcSize* function for more details on their behavior.

HLA high-level calling sequence examples:

```
filePtr.puth8Size( byteVar, width, padChar );
```

**<object>.putw( w:word );**

This procedure writes the value of *w* to the file using exactly four hexadecimal digits (including leading zeros if necessary).

HLA high-level calling sequence examples:

```
filePtr.putw( wordVar );

// If the word is in a register (AX):

MyOutputFile.putw( ax );
```

**<object>.puth16( w:word );**

This procedure writes the value of *w* to the file using 1-4 hexadecimal digits (the minimum necessary).

HLA high-level calling sequence examples:

```
filePtr.puth16( wordVar );

// If the word is in a register (AX):
```

```
MyOutputFile.puth16( ax );
```

**<object>.puth16Size( w:word; width:dword; fill:char )**

This procedure writes the value of *w* to the file using a minimum field *width* and a *fill* character. See the discussion of *width* and *fill* in the description of the <object>.putcSize function for more details on their behavior.

HLA high-level calling sequence examples:

```
filePtr.puth16Size( wordVar, width, padChar );
```

**<object>.putd( dw:dword );**

This procedure writes the value of *d* to the file using exactly eight hexadecimal digits (including leading zeros if necessary). If the stdlib global underscores value (see the conversions module for details) contains true, then this function will also print an underscore between the fourth and fifth digits.

HLA high-level calling sequence examples:

```
filePtr.putd(dwordVar );

// If the dword value is in a register (EAX):

MyOutputFile.putd( eax );
```

**<object>.puth32( dw:dword );**

This procedure writes the value of *d* to the file using the minimum number of hexadecimal required. If the stdlib global underscores value (see the conversions module for details) contains true, then this function will also print an underscore between the fourth and fifth digits (if there are at least five digits in the number).

HLA high-level calling sequence examples:

```
filePtr.puth32( dwordVar );

// If the dword is in a register (EAX):

MyOutputFile.puth32( eax );
```

**<object>.puth32Size( d:dword; width:dword; fill:char )**

This procedure writes the value of *d* to the file using a minimum field *width* and a *fill* character. See the discussion of *width* and *fill* in the description of the <object>.putcSize function for more details on their behavior.

HLA high-level calling sequence examples:

```

filePtr.puth32Size( dwordVar, width, ' ' );

// If the dword is in a register (EAX):

MyOutputFile.puth32Size( eax, width, cl );

```

**<object>.putq( q:qword );**

This procedure writes the value of *q* to the file using exactly 16 hexadecimal digits (including leading zeros if necessary). If the `stdlib` global `underscores` value (see the `conversions` module for details) contains true, then this function will also print an underscore between each group of four digits.

HLA high-level calling sequence example:

```
filePtr.putq( qwordVar );
```

**<object>.puth64( q:qword );**

This procedure writes the value of *q* to the file using 1-16 hexadecimal digits (the minimum necessary). If the `stdlib` global `underscores` value (see the `conversions` module for details) contains true, then this function will also print an underscore between each group of four digits.

HLA high-level calling sequence example:

```
MyOutputFile.puth64( qwordVar );
```

**<object>.puth64Size( q:qword; width:dword; fill:char )**

This procedure writes the value of *q* to the file using a minimum field *width* and a *fill* character. See the discussion of *width* and *fill* in the description of the `<object>.putcSize` function for more details on their behavior.

HLA high-level calling sequence example:

```
MyOutputFile.puth64Size( qwordVar, width, ' ' );
```

**<object>.puttb( tb:tbyte )**

This procedure writes the value of *tb* to the file using exactly 20 hexadecimal digits (including leading zeros if necessary and an intervening underscores if underscore output is enabled).

HLA high-level calling sequence examples:

```
filePtr.puttb( tbyteVar );
```

**<object>.puth80( tb:tbyte )**

This procedure writes the value of *tb* to the file using 1-20 hexadecimal digits (the minimum necessary) and an intervening underscores if underscore output is enabled).

HLA high-level calling sequence examples:

```
filePtr.puth80( tbyteVar );
```

**<object>.puth80Size( tb:tbyte; width:dword; fill:char )**

This procedure writes the value of *tb* to the file using a minimum field *width* and a *fill* character. See the discussion of *width* and *fill* in the description of the *<object>.putcSize* function for more details on their behavior.

HLA high-level calling sequence examples:

```
filePtr.puth80Size( tbyteVar, width, ' ' );
```

**<object>.putl( l:lword )**

This procedure writes the value of *l* to the file using exactly 32 hexadecimal digits (including leading zeros if necessary and an intervening underscores if underscore output is enabled).

HLA high-level calling sequence examples:

```
MyOutputFile.putl( lwordVar );
```

**<object>.puth128( l:lword )**

This procedure writes the value of *l* to the file using 1-32 hexadecimal digits (the minimum necessary) and an intervening underscores if underscore output is enabled).

HLA high-level calling sequence examples:

```
filePtr.puth128( lwordVar );
```

**<object>.puth128Size( l:lword; width:dword; fill:char )**

This procedure writes the value of *l* to the file using a minimum field *width* and a *fill* character. See the discussion of *width* and *fill* in the description of the *<object>.putcSize* function for more details on their behavior.

HLA high-level calling sequence examples:

```
MyOutputFile.puth128Size( tbyteVar, width, ' ' );
```

## 14.9 Signed Integer Numeric Output

The following file output routines all assume that you've opened the *<object>* file variable via a call to *<object>.open* and you've successfully opened the file for output.

These routines convert signed integer values to string format and write that string to the *filevar* file. The `<object>.putxxxSize` functions contain *width* and *fill* parameters that let you specify the minimum field width when outputting a value.

If the absolute value of *width* is greater than the number of print positions the value requires, then these functions output *width* characters to the output file. If *width* is non-negative, then these functions right-justify the value in the output field; if value is negative, then these functions left-justify the value in the output field.

These functions print the *fill* character as the padding value for the extra print positions.

```
xxxSize( value, width, fill );
```

Assuming “value” requires five print positions, “width” is eight, and fill is “f” then the `xxxSize` functions produce the string

f	f	f	V	A	L	U	E
---	---	---	---	---	---	---	---

Assuming “value” requires five print positions, “width” is minus eight, and fill is “f” then the `xxxSize` functions produce the string

V	A	L	U	E	f	f	f
---	---	---	---	---	---	---	---

```
<object>.puti8 ( b:byte );
```

This function converts the eight-bit signed integer you pass as a parameter to a string and writes this string to the file using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
filePtr.puti8( byteVar );
```

```
// If the character is in a register (AL):
```

```
MyOutputFile.puti8( al );
```

```
<object>.puti8Size ( b:byte; width:int32; fill:char );
```

This function writes the eight-bit signed integer value you pass to the specified output file using the *width* and *fill* values as specified above.

HLA high-level calling sequence examples:

```
filePtr.puti8Size( byteVar, width, padChar );
```

```
<object>.puti16( w:word );
```

This function converts the 16-bit signed integer you pass as a parameter to a string and writes this string to the file using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
MyOutputFile.puti16( wordVar );
```

```
// If the word is in a register (AX):
```

```
filePtr.puti16( ax );
```



**<object>.puti16Size( w:word; width:int32; fill:char );**

This function writes the 16-bit signed integer value you pass to the specified output file using the *width* and *fill* values as specified above.

HLA high-level calling sequence examples:

```
filePtr.puti16Size( wordVar, width, padChar );
```

**<object>.puti32( d:dword );**

This function converts the 32-bit signed integer you pass as a parameter to a string and writes this string to the file using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
filePtr.puti32( dwordVar );
```

```
// If the dword is in a register (EAX):
```

```
MyOutputFile.puti32( eax );
```

**<object>.puti32Size( d:dword; width:int32; fill:char );**

This function writes the 32-bit value you pass as a signed integer to the specified output file using the *width* and *fill* values as specified above.

HLA high-level calling sequence examples:

```
MyOutputFile.puti32Size( dwordVar, width, ' ' );
```

```
// If the dword is in a register (EAX):
```

```
filePtr.puti32Size( eax, width, cl );
```

**<object>.puti64( q:qword );**

This function converts the 64-bit signed integer you pass as a parameter to a string and writes this string to the file using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
MyOutputFile.puti64( qwordVar );
```

```
<object>.puti64Size( q:qword; width:int32; fill:char );
```

This function writes the 64-bit value you pass as a signed integer to the specified output file using the *width* and *fill* values as specified above.

HLA high-level calling sequence examples:

```
filePtr.puti64Size( qwordVar, width, ' ' );
```

```
<object>.puti128( l:lword );
```

This function converts the 128-bit signed integer you pass as a parameter to a string and writes this string to the file using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
MyOutputFile.puti128( lwordVar );
```

```
<object>.puti128Size( l:lword; width:int32; fill:char );
```

This function writes the 128-bit value you pass as a signed integer to the specified output file using the *width* and *fill* values as specified above.

HLA high-level calling sequence examples:

```
filePtr.puti128Size( lwordVar, width, ' ' );
```

## 14.10 Unsigned Integer Numeric Output

These routines convert unsigned integer values to string format and write that string to the file. The *<object>.putxxxSize* functions contain *width* and *fill* parameters that let you specify the minimum field width when outputting a value.

If the absolute value of *width* is greater than the number of print positions the value requires, then these functions output *width* characters to the output file. If *width* is non-negative, then these functions right-justify the value in the output field; if value is negative, then these functions left-justify the value in the output field.

These functions print the *fill* character as the padding value for the extra print positions.

```
<object>.putu8 ( b:byte )
```

This function converts the eight-bit unsigned integer you pass as a parameter to a string and writes this string to the file using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
filePtr.putu8( byteVar );
```

```
// If the character is in a register (AL):
```

```
MyOutputFile.putu8( al );
```

**<object>.putu8size( b:byte; width:int32; fill:char )**

This function writes the unsigned eight-bit value you pass to the specified output file using the *width* and *fill* values as specified above.

HLA high-level calling sequence examples:

```
filePtr.putu8Size( byteVar, width, padChar );
```

**<object>.putu16( w:word )**

This function converts the 16-bit unsigned integer you pass as a parameter to a string and writes this string to the file using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
filePtr.putu16( wordVar );
```

```
// If the word is in a register (AX):
```

```
MyOutputFile.putu16( ax );
```

**<object>.putu16size( w:word; width:int32; fill:char )**

This function writes the unsigned 16-bit value you pass to the specified output file using the *width* and *fill* values as specified above.

HLA high-level calling sequence examples:

```
filePtr.putu16Size( wordVar, width, padChar );
```

**<object>.putu32( d:dword )**

This function converts the 32-bit unsigned integer you pass as a parameter to a string and writes this string to the file using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
filePtr.putu32( dwordVar );
```

```
// If the dword is in a register (EAX):
```

```
MyOutputFile.putu32( eax );
```

**<object>.putu32Size( d:dword; width:int32; fill:char )**

This function writes the unsigned 32-bit value you pass to the specified output file using the *width* and *fill* values as specified above.

HLA high-level calling sequence examples:

```
MyOutputFile.putu32Size( dwordVar, width, ' ' );
```

```
// If the dword is in a register (EAX):
```

```
filePtr.putu32Size( eax, width, cl );
```

#### **<object>.putu64( q:qword )**

This function converts the 64-bit unsigned integer you pass as a parameter to a string and writes this string to the file using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
filePtr.putu64( qwordVar );
```

#### **<object>.putu64Size( q:qword; width:int32; fill:char );**

This function writes the unsigned 64-bit value you pass to the specified output file using the *width* and *fill* values as specified above.

HLA high-level calling sequence examples:

```
MyOutputFile.putu64Size( qwordVar, width, ' ' );
```

#### **<object>.putu128( l:lword )**

This function converts the 128-bit unsigned integer you pass as a parameter to a string and writes this string to the file using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
filePtr.putu128( lwordVar );
```

#### **<object>.putu128Size( l:lword; width:int32; fill:char );**

This function writes the unsigned 128-bit value you pass to the specified output file using the *width* and *fill* values as specified above.

HLA high-level calling sequence examples:

```
MyOutputFile.putu128Size( lwordVar, width, ' ' );
```

## 14.11 Floating-Point Numeric Output Using Scientific Notation

The floating point numeric output routines translate the three different binary floating point formats to their string representation and then write this string to the file that filevar specifies. There are two generic classes of these routines: those that convert their values to exponential/scientific notation and those that convert their string to a decimal form.

The `<object>.pute80`, `<object>.pute64`, and `<object>.pute32` routines convert their values to a string using scientific notation. These three routines each have two parameters: the value to output and the field width of the result. These routines produce a string with the following format:

s	i	.	f	f	f	f	f	E	±	x
---	---	---	---	---	---	---	---	---	---	---

s is a space for positive values, '-' for negative values  
i represents the integer portion of the mantissa  
ffff represents the fractional portion of the mantissa  
x is one or more base-10 exponent digits.

**`<object>.pute32( r:real32; width:uns32 )`**

This function writes the 32-bit single precision floating point value passed in *r* to the file using scientific/exponential notation. This procedure prints the value using *width* print positions in the file. *width* should have a minimum value of five for real numbers in the range 1e-9..1e+9 and a minimum value of six for all other values. Note that 32-bit extended precision floating point values support about 6-7 significant digits. So a *width* value that yields more than seven mantissa digits will produce garbage output in the low order digits of the number.

HLA high-level calling sequence examples:

```
MyOutputFile.pute32( r32Var, width );

// If the real32 value is in an FPU register (ST0):

var
    r32Temp:real32;
    .
    .
    .
fstp( r32Temp );
filePtr.pute32( r32Temp, 12 );
```

**`<object>.pute64( r:real64; width:uns32 )`**

This function writes the 64-bit double precision floating point value passed in *r* to the file using scientific/exponential notation. This procedure prints the value using *width* print positions in the file. *width* should have a minimum value of five for real numbers in the range 1e-9..1e+9 and a minimum value of six for all other values. Note that 64-bit double precision floating point values support about 15 significant digits. So a *width* value that yields more than 15 mantissa digits will produce garbage output in the low order digits of the number.

HLA high-level calling sequence examples:

```
filePtr.pute64( r64Var, width );

// If the real64 value is in an FPU register (ST0):
```

```

var
    r64Temp:real64;
    .
    .
    .
fstp( r64Temp );
filePtr.put64( r64Temp, 12 );

```

**<object>.pute80( r:real80; width:uns32 )**

This function writes the 80-bit extended precision floating point value passed in *r* to the file using scientific/exponential notation. This procedure prints the value using *width* print positions in the file. *width* should have a minimum value of five for real numbers in the range 1e-9..1e+9 and a minimum value of six for all other values. Note that 80-bit extended precision floating point values support about 18 significant digits. So a *width* value that yields more than 18 mantissa digits will produce garbage output in the low order digits of the number.

HLA high-level calling sequence examples:

```

filePtr.put80( r80Var, width );

// If the real80 value is in an FPU register (ST0):

var
    r80Temp:real80;
    .
    .
    .
fstp( r80Temp );
MyOutputFile.put80( r80Temp, 12 );

```

## 14.12 Floating-Point Numeric Output Using Decimal Notation

Although scientific (exponential) notation is the most general display format for real numbers, real numbers you display in this format are very difficult to read. Therefore, the HLA file class module also provides a set of functions that output real values using the decimal representation. Although you cannot (practically) use these decimal output routines for all real values, they are applicable to a wide variety of common numbers you will use in your programs.

These functions come in two varieties. The first variety requires four parameters: the real value to convert, the width of the converted value, the number of digit positions to the right of the decimal point, and a padding character. The second variety only requires the first three parameters and assumes the padding character is a space. These functions write their values using the following string format:

s	i	i	i	.	f	f	f	f	f	f
---	---	---	---	---	---	---	---	---	---	---

s is a space for positive values, '-' for negative values  
i represents the integer portion of the mantissa  
ffff represents the fractional portion of the mantissa

**<object>.putr32( r:real32; width:uns32; decpts:uns32; fill:char )**

This procedure writes a 32-bit single precision floating point value to the file as a string. The string consumes exactly *width* characters in the output file. If the numeric output, using the specified number of positions to the right of the decimal point, is sufficiently small that the string representation would be less than *width* characters, then this procedure uses the *fill* value as the padding character to fill the output with *width* characters.

HLA high-level calling sequence examples:

```
filePtr.putr32( r32Var, width, decpts, fill );
filePtr.putr32( r32Var, 10, 2, '*' );

// If the real32 value is in an FPU register (ST0):

var
    r32Temp:real32;
    .
    .
    .
fstp( r32Temp );
filePtr.putr32( r32Temp, 12, 2, al );
```

**<object>.putr64( r:real64; width:uns32; decpts:uns32; fill:char )**

This procedure writes a 64-bit double precision floating point value to the file as a string. The string consumes exactly *width* characters in the output file. If the numeric output, using the specified number of positions to the right of the decimal point, is sufficiently small that the string representation would be less than *width* characters, then this procedure uses the value of *fill* as the padding character to fill the output with *width* characters.

HLA high-level calling sequence examples:

```
MyOutputFile.putr64( r64Var, width, decpts, fill );
MyOutputFile.putr64( r64Var, 10, 2, '*' );

// If the real64 value is in an FPU register (ST0):

var
    r64Temp:real64;
    .
    .
    .
fstp( r64Temp );
MyOutputFile.putr64( r64Temp, 12, 2, al );
```

**<object>.putr80( r:real80; width:uns32; decpts:uns32; fill:char )**

This procedure writes an 80-bit extended precision floating point value to the file as a string. The string consumes exactly *width* characters in the output file. If the numeric output, using the specified number of positions to the right of the decimal point, is sufficiently small that the string representation would be less than *width* characters, then this procedure uses the value of *fill* as the padding character to fill the output with *width* characters.

HLA high-level calling sequence examples:

```

filePtr.putr80( r80Var, width, decpts, fill );
filePtr.putr80( r80Var, 10, 2, '*' );

// If the real80 value is in an FPU register (ST0):

var
    r80Temp:real80;
.
.
.
fstp( r80Temp );
filePtr.putr80( r80Temp, 12, 2, al );

```

## 14.13 Generic File Output

**<object>.put( parameter\_list )**

*<object>.put* is a macro that automatically invokes an appropriate *<object>* output routine based on the type of the parameter(s) you pass it. This is a very convenient output routine and is probably the file class output call you will use most often in your programs. Keep in mind that this macro is not a single function call; instead, HLA translates this macro into a sequence of calls to procedures like *<object>.puti32*, *<object>.puts*, etc.

*<object>.put* is a macro that provides a flexible syntax for outputting data to the standard output device. This macro allows a variable number of parameters. For each parameter present in the list, *<object>.put* will call the appropriate routine to emit that data, according to the type of the parameter. Parameters may be constants, registers, or memory locations. You must separate each macro parameter with a comma.

Here is an example of a typical invocation of an *<object>.put* :

```
<object>.put( "I=", i, " j=", j, nl );
```

The above is roughly equivalent to

```

<object>.puts( "I=" );
<object>.puti32( i );
<object>.puts( " j=" );
<object>.puti32( j );
<object>.newline();

```

This assumes, of course, that *i* and *j* are *int32* variables.

The *<object>.put* macro also lets you specify the minimum field width for each parameter you specify. To print a value using a minimum field width, follow the object you wish to print with a colon and the value of the minimum field width. The previous example, using field widths, could look like the following:

```
<object>.put( "I=", i:2, " j=", j:5, nl );
```

Although this example used the literal decimal constants two and five for the field widths, keep in mind that register values and memory value (integers, anyway) are perfectly legal here.

For floating point numbers you wish to display in decimal form, you can specify both the minimum field width and the number of digits to print to the right of the decimal point by using the following syntax:

```
<object>.put( "Real value is ", f:10:3, nl );
```

The *<object>.put* macro can handle all the basic primitive types, including boolean, unsigned (8, 16, 32, 64, 128), signed (8, 16, 32, 64, 128), character, character set, real (32, 64, 80), string, and hexadecimal (byte, word, dword, qword, tbyte, lword).

If you specify a class variable (object) and that class defines a *toString* method, the *<object>.put* macro will call the associated *toString* method and output that string to the file. Note that the *toString* method must dynamically allocate storage for the string by calling *str.alloc*. This is because *<object>.put* will call *str.free* on the string once it outputs the string.



There is a known "design flaw" in the `<object>.put` macro. You cannot use it to print HLA intermediate variables (i.e., non-local VAR objects). The problem is that HLA's syntax for non-local accesses takes the form "reg32:varname" and `<object>.put` cannot determine if you want to print `reg32` using `varname` print positions versus simply printing the non-local `varname` object. If you want to display non-local variables you must copy the non-local object into a register, a static variable, or a local variable prior to using `<object>.put` to print it. Of course, there is no problem using the other `<object>.putXXXX` functions to display non-local VAR objects, so you can use those as well.

**Important(!)**, don't forget that method calls (e.g., the routines that `<object>.put` translates into) modify the values in the ESI and EDI registers. Therefore, it never makes any sense to attempt to print the values of ESI and EDI within the parameter list. All you will wind up doing is printing the address of the file variable (ESI) or the address of its virtual method table (EDI). If you need to write these two values to a file, move them to another register or a memory location first.

## 14.14 Generic File Input

The following file input routines behave just like their standard input and file input counterparts (unless otherwise noted):

```
<object>.read( var buffer:var; count:dword )
```

This function reads `count` bytes from the file and stores them into memory starting with the first byte of the `buffer` variable. This routine does not do any range checking. It is your responsibility to ensure that `buffer` is large enough to hold the data read.

Note: Notice that the `buffer` parameter is an untyped reference parameter. Untyped reference parameters have special properties, so be sure to read the chapter on "Passing Parameters to Standard Library Routines" (parmpassing.rtf) if you are not absolutely sure you understand how untyped reference parameters operate.

HLA high-level calling sequence examples:

```
MyInputFile.read( buffer, count );
MyInputFile.read( [eax], 1024 );
```

```
<object>.readln;
```

This function reads and discards all characters up to and including the newline sequence in the file.

HLA high-level calling sequence examples:

```
filePtr.readLn();
```

## 14.15 Character and String Input

The following functions read character data from an input file. Note that HLA's file class module does not provide the ability to read character set data directly from the user. However, you can always read a string and then convert that string to a character set using the appropriate function in the `cset` module.

```
<object>.getc; @returns( "al" );
```

This function reads a single character from the file and returns that character in the AL register. This function assumes that the file you've opened is a text file. Note that `<object>.getc` does not return the end of line sequence as part of the input stream. Use the `<object>..eoln` function to determine when you've reached the end of a line of text. Because `<object>..getc` preprocesses the text file (removing end of line sequences) you should not use it to read binary data, use it only to read text files.

HLA high-level calling sequence examples:

```
filePtr.getc();
```

```
mov( al, charVar );
```

```
<object>.gets( s:string );
```

This function reads a sequence of characters from the current file position through to the next end of line sequence and stores these characters (without the end of line sequence) into the string variable you pass as a parameter. Before calling this routine, you must allocate sufficient storage for the string. If *<object>.gets* attempts to read a larger string than the string's *MaxLen* value, *<object>.gets* raises a string overflow exception.

Note that this function does not store the end of line sequence into the string, though it does consume the end of line sequence. The next character a file class function will read from the file will be the first character of the following line.

If the current file position is at the end of some line of text, then *<object>.gets* consumes the end of line and stores the empty string into the *s* parameter.

HLA high-level calling sequence examples:

```
filePtr.gets( inputStr );
filePtr.gets( eax ); // EAX contains string value
```

```
<object>.a_gets; @returns( "eax" );
```

Like *<object>.gets*, this function also reads a string from the file. However, rather than storing the string data into a string you supply, this function allocates storage for the string on the heap and returns a pointer to this string in the EAX register. You code should call *str.free* to release this storage when you're done with the string data.

The *<object>.a\_gets* function imposes a line length limit of 4,096 characters. If this is a problem, you should modify the source code for this function to raise the limit. This function raises an exception if you attempt to read a line longer than this internal limit.

HLA high-level calling sequence examples:

```
MyInputFile.a_gets();
mov( eax, inputStr );
```

## 14.16 Signed Integer Input

The functions in this group read numeric values from the file using a signed decimal integer format. These functions read the string data, translate it to numeric form, and return that numeric data in an appropriate location.

```
<object>.geti8; @returns( "al" );
```

This function reads a signed eight-bit decimal integer in the range -128..+127 from the file. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by an optional minus sign and a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The *<object>.geti8* function raises an appropriate exception if the input violates any of these rules or the value is outside the range -128..+127. This function returns the binary form of the integer in the AL register.

HLA high-level calling sequence examples:

```
filePtr.geti8();
```

```
mov( al, i8Var );
```

```
<object>.geti16; @returns( "ax" );
```

This function reads a signed 16-bit decimal integer in the range -32768..+32767 from the file. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by an optional minus sign and a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The *<object>.geti16* function raises an appropriate exception if the input violates any of these rules or the value is outside the range -32768..+32767. This function returns the binary form of the integer in the AX register.

HLA high-level calling sequence examples:

```
filePtr.geti16();
mov( ax, i16Var );
```

```
<object>.geti32; @returns( "eax" );
```

This function reads a signed 32-bit decimal integer in the (approximate) range  $\pm 2$  Billion from the file. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by an optional minus sign and a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The *<object>.geti32* function raises an appropriate exception if the input violates any of these rules or the value is outside the range plus or minus two billion. This function returns the binary form of the integer in the EAX register.

HLA high-level calling sequence examples:

```
filePtr.geti32();
mov( eax, i32Var );
```

```
<object>.geti64; @returns( "edx:eax" );
```

This function reads a signed 64-bit decimal integer from the file. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by an optional minus sign and a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The *<object>.geti64* function raises an appropriate exception if the input violates any of these rules or the value is outside the range of a 64-bit signed integer. This function returns the 64-bit result in the EDX:EAX register pair (it returns the H.O. dword in EDX and the L.O. dword in EAX).

HLA high-level calling sequence examples:

```
filePtr.geti64();
mov( edx, (type dword i64Var[4]) );
mov( eax, (type dword i64Var[0]) );
```

```
<object>.geti128( var l:lword );
```

This function reads a signed 128-bit decimal integer from the file. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by an optional minus sign and a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The *<object>.geti128* function raises an appropriate exception if the input violates any of these rules or the value is outside the range of a 128-bit signed integer. This function stores the 128-bit result into the *lword* you pass as a reference parameter.

HLA high-level calling sequence examples:

```
filePtr.geti128( lwordVar );
```

## 14.17 Unsigned Integer Input

The functions in this group read numeric values from the file using a signed decimal integer format. These functions read the string data, translate it to numeric form, and return that numeric data in an appropriate location.

```
<object>.getu8; @returns( "al" );
```

This function reads an unsigned eight-bit decimal integer in the range 0..+255 from the file. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The *<object>.getu8* function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..255. This function returns the binary form of the integer in the AL register.

HLA high-level calling sequence examples:

```
MyInputFile.getu8();
mov( al, u8Var );
```

```
<object>.getu16; @returns( "ax" );
```

This function reads an unsigned 16-bit decimal integer in the range 0..+65535 from the file. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The *<object>.getu16* function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..65535. This function returns the binary form of the integer in the AX register.

HLA high-level calling sequence examples:

```
filePtr.getu16();
mov( ax, u16Var );
```

```
<object>.getu32; @returns( "eax" );
```

This function reads an unsigned 32-bit decimal integer in the range 0..+4,294,967,295 from the file. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The *<object>.getu32* function raises an appropriate exception if the input violates any of these rules

or the value is outside the range 0..4,294,967,295. This function returns the binary form of the integer in the EAX register.

HLA high-level calling sequence examples:

```
filePtr.getu32();
mov( eax, u32Var );
```

**<object>.getu64; @returns( "edx:eax" );**

This function reads an unsigned 64-bit decimal integer from the file. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The *<object>.getu64* function raises an appropriate exception if the input violates any of these rules or the value is outside the range  $0..2^{64}-1$ . This function returns the binary form of the integer in EDX:EAX register pair (EDX contains the H.O. dword, EAX holds the L.O. dword).

HLA high-level calling sequence examples:

```
filePtr.getu32();
mov( eax, (type dword u64Var) );
mov( edx, (type dword u64Var[4]) );
```

**<object>.getu128( var l:lword );**

This function reads an unsigned 128-bit decimal integer from the file. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The *<object>.getu64* function raises an appropriate exception if the input violates any of these rules or the value is outside the range  $0..2^{128}-1$ . This function returns the binary form of the integer in the lword parameter you pass by reference.

HLA high-level calling sequence examples:

```
fileio.getu128( u128Var );
```

## 14.18 Hexadecimal Input

**<object>.geth8; @returns( "al" );**

This function reads an eight-bit hexadecimal integer in the range 0..\$FF from the file. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "\$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The *<object>.geth8* function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..\$FF. This function returns the binary form of the value in the AL register.

HLA high-level calling sequence examples:

```
filePtr.geth8();
```

```
mov( al, h8Var );
```

```
<object>.geth16; @returns( "ax" );
```

This function reads a 16-bit hexadecimal integer in the range 0..\$FFFF from the file. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "\$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The *<object>.geth16* function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..\$FFFF. This function returns the binary form of the value in the AX register.

HLA high-level calling sequence examples:

```
MyInputFile.geth16();
mov( ax, h16Var );
```

```
<object>.geth32; @returns( "eax" );
```

This function reads a 32-bit hexadecimal integer in the range 0..\$FFFF\_FFFF from the file. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "\$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The *<object>.geth32* function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..\$FFFF\_FFFF. This function returns the binary form of the value in the EAX register.

HLA high-level calling sequence examples:

```
filePtr.geth32();
mov( eax, h32Var );
```

```
<object>.geth64; @returns( "edx:eax" );
```

This function reads a 64-bit hexadecimal integer in the range 0..\$FFFF\_FFFF\_FFFF\_FFFF from the file. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "\$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The *<object>.geth64* function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..\$FFFF\_FFFF\_FFFF\_FFFF. This function returns the 64-bit result in the EDX:EAX register pair (EDX contains the H.O. dword, EAX contains the L.O. dword).

HLA high-level calling sequence examples:

```
MyInputFile.geth64();
mov( edx, (type dword h64Var[4]) );
mov( eax, (type dword h64Var[0]) );
```

```
<object>.geth128( var l:lword );
```

This function reads a 128-bit hexadecimal integer in the range 0..\$FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF from the file. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "\$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The *<object>.getq* function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..\$FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF. This function stores the 128-bit result into the variable you pass as a reference parameter.

HLA high-level calling sequence examples:

```
filePtr.geth128( lwordVar );
```

## 14.19 Floating-Point Input

```
<object>.getf; @returns( "st0" );
```

This function reads an 80-bit floating point value in either decimal or scientific from the file and leaves the result sitting on the FPU stack. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by an optional minus sign and a sequence of characters that represent a floating point value. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. This function raises an appropriate exception if an error occurs.

HLA high-level calling sequence examples:

```
filePtr.getf();
fstp( fpVar );
```

## 14.20 Generic File Input

```
<object>.get( List_of_items_to_read );
```

This is a macro that allows you to specify a list of variable names as parameters. The *<object>.get* macro reads an input value for each item in the list and stores the resulting value in each corresponding variable. This macro determines the type of each variable that you pass it and emits a call to the appropriate *<object>.getxxx* function to read the actual value. As an example, consider the following call to *<object>.get*:

```
filePtr.get( i32, charVar, u16, strVar );
```

The macro invocation above expands into the following:

```
push( eax );
filePtr.geti32( i32 );
filePtr.getc();
mov( al, charVar );
filePtr.geti16();
mov( ax, u16 );
filePtr.gets( strVar );
pop( eax );
```

Notice that `<object>.get` preserves the value in the EAX register even though various `<object>.getxxx` functions use this register. Note that `<object>.get` automatically handles the case where you specify EAX as an input variable and writes the value to `[esp]` so that it properly modifies EAX upon completion of the macro expansion.

Note that `<object>.get` only supports eight-, sixteen-, and thirty-two bit integer input. If you need to read 64-bit or 128-bit values, you must use the appropriate `<object>.getx64` or `<object>.getx128` function to achieve this.



## 15 The File I/O Module (fileio.hhf)

This unit contains routines that read data from and write data to files. The fileio functions can be broken down into four generic categories: general functions that open and close files, file position functions that get or set the current file position (or test the file position), output functions that write data to a file, and input functions that read data from a file.

Note to stdlib v1.x users: Several routines originally found in the fileio package have been moved to the new filesys package. The affected routines did not operate on file data, but on the file system itself. Examples include file deletion, get working directory, and change directory. Please see the filesys.rtf documentation for a description of those routines.

Note: be sure to read the chapter on "Passing Parameters to Standard Library Routines" (parmpassing.rtf) before reading this chapter.

**Note about stack diagrams:** this documentation includes stack diagrams for those functions that pass parameters on the stack. To conserve space, this documentation does not include a stack diagram for any function that does not pass data on the stack (that is, only a return address appears on the stack).

**A Note About the FPU:** The Standard Library code makes occasional use of the FPU, particularly when converting between real and string formats and when computing certain mathematical functions. You should exercise caution when using MMX instructions in a program that makes use of the Standard Library. In particular, you should ensure that you are always in FPU mode (by executing an EMMS instruction) after you are finished using MMX instructions. Better yet, you should avoid the MMX instruction set altogether and use the improved SSE instruction set that accomplishes the same tasks (and doesn't disturb the FPU).

### 15.1 Conversion Format Control

The fileio output functions that convert numeric values to hexadecimal, unsigned decimal, and signed decimal output provide the ability to inject underscores between groups of three (decimal) or four (hexadecimal) digits to make it easier to read large numbers. You enable and disable underscore output using the conv.setUnderscores and conv.getUnderscores functions. Please refer to their documentation in the conv.rtf file for more details.

When reading numeric data from a text file, the fileio functions use an internal delimiters character set to determine which characters may legally end a sequence of numeric digits. You can change the complexion of this character set using the conv.getDelimiters and conv.setDelimiters functions. Please refer to their documentation in the conv.rtf file for more details.

When converting numeric values to string form for output, the fileio routines call the conversion functions found in the conv (conversions) module. For detailed information on the actual conversions, please consult the conv.rtf document.

### 15.2 General File I/O Functions

Here are the file output routines provided by the HLA fileio unit:

Note: fileio.open is part of the os\_fileio module in v2.0 of the HLA stdlib. This function has not been updated yet and the semantics may change during the conversion to v2.0.

```
fileio.open( FileName: string; Access:dword ); @returns( "eax" );
```

The fileio.open routine opens the file by the specified name. The Access parameter is one of the following:

- fileio.r
- fileio.w
- fileio.rw
- fileio.a

The fileio.r constant tells HLA to open the file for read-only access. The fileio.w constant tells HLA to open the file for writing. Using the fileio.rw constant tells fileio.open to open the file for reading and writing. The fileio.a option tells the fileio.open function to open the file for writing and append all written data to the end of the file.

This routine raise an exception if there is a problem opening the file (e.g., the file does not exist). If the file is successfully opened, this function returns the file handle in the EAX register.

HLA high-level calling sequence examples:

```
fileio.open( "myfile.txt", fileio.r );
mov( eax, fileHandle );

// Note: the Access parameter is almost always a constant in
// calls to fileio.open. However, if you want to pass a variable
// value or a register value in this parameter, you may certainly
// do so:

fileio.open( filenameStr, accessVarByte );
mov( eax, fileHandle );

fileio.open( someStr, al );
mov( eax, fileHandle );
```

HLA low-level calling sequence examples:

```
// Constant Access value:

push( filenameStr );
pushd( fileio.r );
call fileio.open;
mov( eax, fileHandle );

// Access value in register (AL in this example)

push( filenameStr );
push( eax );
call fileio.open;
mov( eax, fileHandle );

// Access value is in a (byte) variable

push( filenameStr );
push( (type dword accessValue)); //Not always safe!
call fileio.open;
mov( eax, fileHandle );

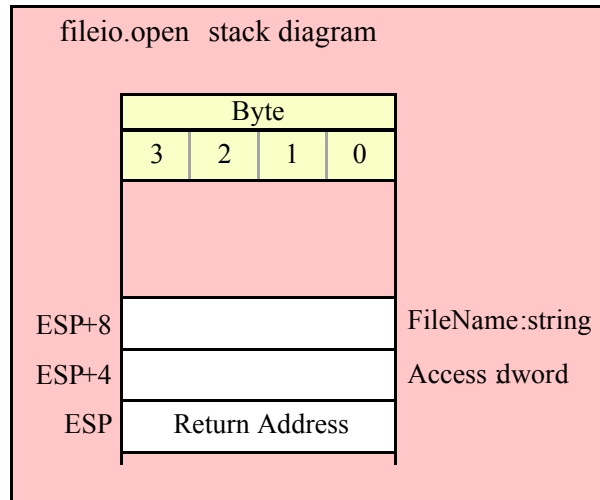
// Solution if accessing accessValue as a dword
// might cause a memory access error (last three
// bytes on a 4K page in memory, etc.):

push( filenameStr );
sub( 4, esp );
push( eax );
movzx( accessValue, eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.open;
mov( eax, fileHandle );

// Note: If you want to use a string literal, the best solution is
// to create a string object in the readonly section, e.g.,
//
// readonly
//   filenameStr :string := "myfile.txt";
//
// and just use the "filenameStr" object you've created. You may also
```

```
// do the following if you have a register available:
```

```
lea( eax, "myfile.txt" );
push( eax );
pushd( fileio.r );
call fileio.open;
mov( eax, fileHandle );
```



```
fileio.openNew( FileName: string ); @returns( "eax" );
```

This function opens a new file for writing. The single parameter specifies the file's (path) name. This function raises an exception if there is an error opening the file. If the file is opened successfully, this function returns the file handle in the EAX register. If the file already exists, this function will successfully open the file and delete any existing data in the file.

HLA high-level calling sequence examples:

```
fileio.openNew( "myfile.txt" );
mov( eax, fileHandle );
```

```
// If the filename string pointer is in a register (EAX):
```

```
fileio.openNew( eax );
mov( eax, fileHandle );
```

HLA low-level calling sequence examples:

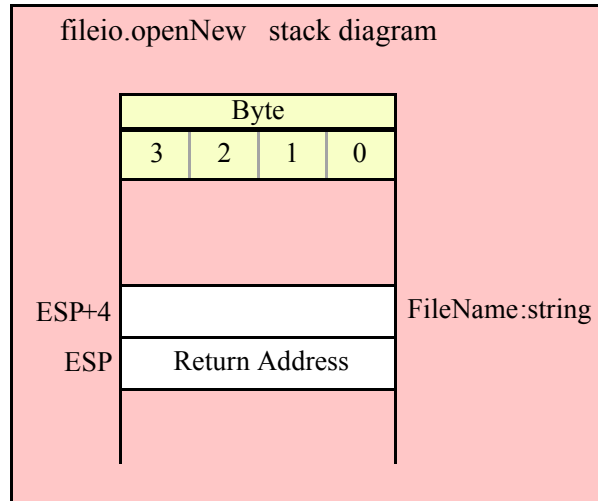
```
push( filenameStr );
call fileio.openNew;
mov( eax, fileHandle );
```

```
// If the string pointer value is in a register (EAX
// in this example):
```

```
push( eax );
call fileio.openNew;
mov( eax, fileHandle );
```

```
// Note: If you want to use a string literal, the best solution is
// to create a string object in the readonly section, e.g.,
//
// readonly
//   filenameStr :string := "myfile.txt";
//
// and just use the "filenameStr" object you've created. You may also
// do the following if you have a register available:
```

```
lea( eax, "myfile.txt" );
push( eax );
call fileio.openNew;
mov( eax, fileHandle );
```



```
fileio.close( Handle:dword );
```

This function closes the file specified by the handle passed as the parameter. You should close all files as soon as you are done using them. Note that successful program termination automatically closes all files, but it is exceeding poor programming practice to rely on the operating system to close any files you've left open. Were the machine to crash, data could be lost; for this reason, you should close all files as soon as you are finished reading and writing data.

HLA high-level calling sequence examples:

```
fileio.close( fileHandle );

// If the file handle is in a register (EAX):

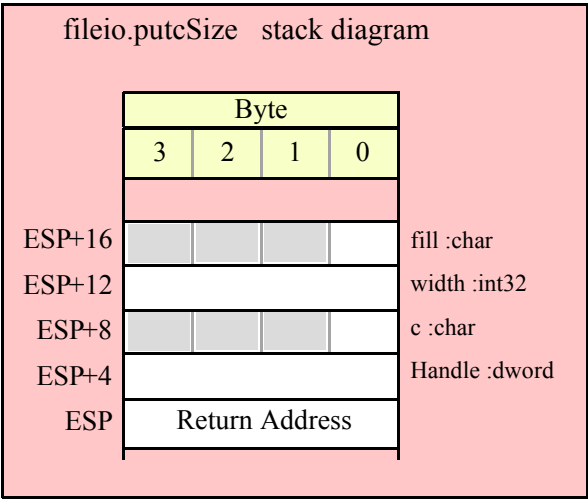
fileio.close( eax );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
call fileio.close;

// If the file handle is in a register (EAX):
```

```
push( eax );
call fileio.close;
```



```
fileio.flush( Handle:dword );
```

This function flushes all pending data to the file (same operation as closing the file, without actually closing it). Note that successful program termination automatically closes all files, but were a crash to occur, some data might be lost. Flushing the file on a periodic basis can help prevent file data loss.

HLA high-level calling sequence examples:

```
fileio.flush( fileHandle );

// If the file handle is in a register (EAX):

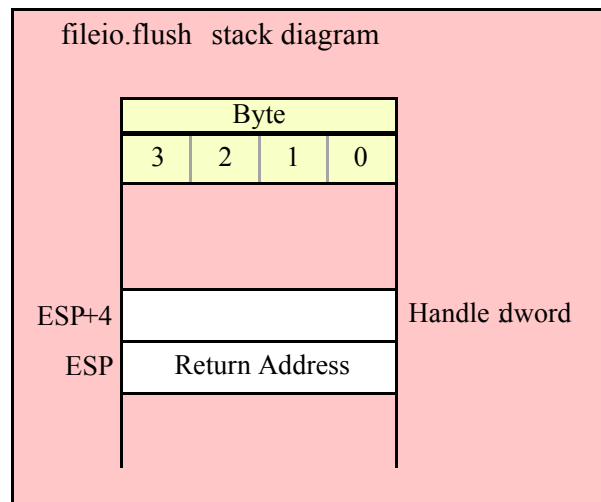
fileio.flush( eax );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
call fileio.flush;

// If the file handle is in a register (EAX):

push( eax );
call fileio.flush;
```



```
fileio.eof( Handle:dword ); @returns( "al" );
```

This function returns true (1) in AL if the specified file is at the end of file. It returns false (0) otherwise. Note that this function actually returns true/false in EAX even though the "returns" value is "AL". So don't count on it preserving the value in AH or the upper 16 bits of EAX.

**Warning:** fileio.eof only functions properly for actual disk files. If you attempt to read data from an interactive device like the system console (keyboard) or a serial port, fileio.eof's behavior is incorrect (it will wind up eating a character from the interactive input stream every time you call it). Unfortunately, neither Windows nor Linux provides a way to test for EOF until after you've actually read a character from the input stream. A better solution, which works fine with both interactive input streams and file data is to use HLA's try..endtry statement to trap and EOF error when it occurs. For example, rather than writing the following:

```
while( !fileio.eof( someHandle )) do
.
.
.
endwhile;
```

You should write the following:

```
try
  forever
    .
    .
    .
  endfor;
  exception( ex.EndOfFile );
endtry;
```

Note: under Windows, fileio.eof always returns false for character device files (e.g., keyboard input) and it returns false for all other non-disk file device types. Note that if the user presses ctrl-Z on the keyboard, fileio.eof will not return true, but the system will return an ex.endOfFile exception. If there is any chance you'll be reading data from a device file rather than a disk file, always use the try..endtry block to test for EOF.

HLA high-level calling sequence examples:

```
while( !fileio.eof( fileHandle ) ) do
  <<something while not at EOF>>
```

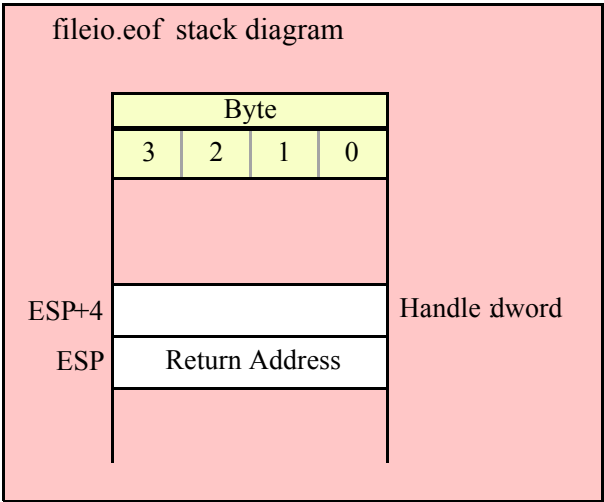
```
endwhile;
```

HLA low-level calling sequence examples:

```
whileNotEOF:
    push( fileHandle );
    call fileio.flush;
    cmp( al, true );
    jne atEOF;
```

```
<< something while not at EOF>>
```

```
    jmp whileNotEOF;
atEOF:
```



```
fileio.rewind( Handle:dword ); @returns( "eax" );
```

The Handle parameter specifies the handle of an open file. This function positions the file pointer to the beginning of the file (file position zero). This function returns the error code in EAX.

HLA high-level calling sequence examples:

```
fileio.rewind( fileHandle );

// If the file handle is in a register (EAX):

fileio.rewind( eax );
```

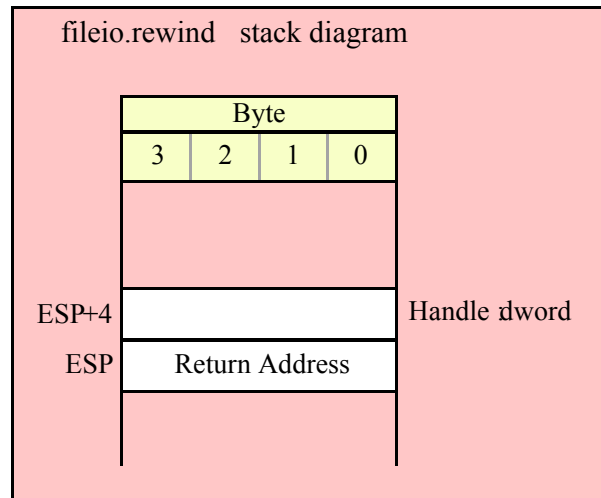
HLA low-level calling sequence examples:

```
push( fileHandle );
call fileio.rewind;

// If the file handle is in a register (EAX):

push( eax );
```

```
call fileio.rewind;
```



```
fileio.append( handle:dword ); @returns( "eax" );
```

This function positions the file pointer of the file specified by the handle parameter to the end of that file. The file should have been opened for writing.

HLA high-level calling sequence examples:

```
fileio.append( fileHandle );
```

```
// If the file handle is in a register (EAX):
```

```
fileio.append( eax );
```

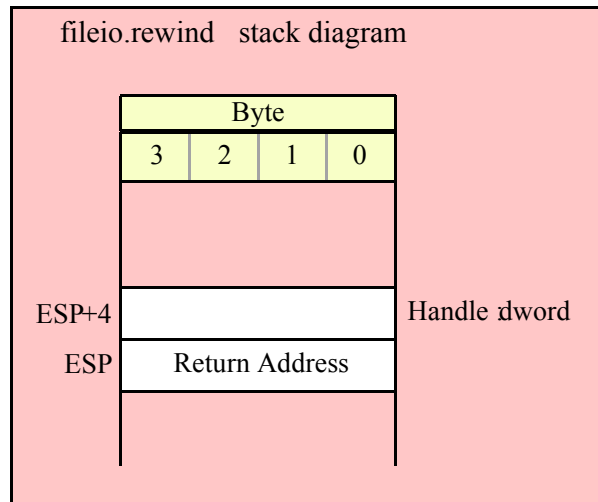
HLA low-level calling sequence examples:

```
push( fileHandle );
call fileio.append;
```

```
// If the file handle is in a register (EAX):
```

```
push( eax );
call fileio.append;
```





```
fileio.position( Handle:dword ); @returns( "eax" );
```

This function returns the file position (in bytes) of the file specified by the handle parameter. It returns the file position offset in the EAX register.

HLA high-level calling sequence examples:

```
fileio.position( fileHandle );
mov( eax, (type dword filePosition));

// If the file handle is in a register (EAX):

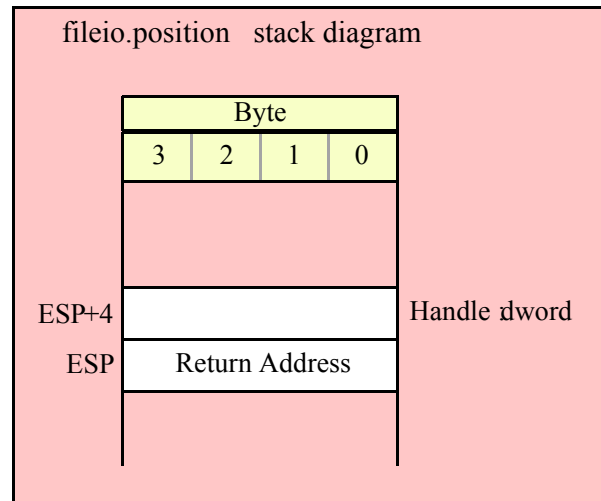
fileio.position( eax );
mov( eax, (type dword filePosition));
```

HLA low-level calling sequence examples:

```
push( fileHandle );
call fileio.position;
mov( eax, (type dword filePosition));

// If the file handle is in a register (EAX):

push( eax );
call fileio.position;
mov( eax, (type dword filePosition));
```



```
fileio.seek( Handle:dword; offset:qword ); @returns( "eax" );
```

This function sets the file position in the file specified by the Handle parameter to the position specified by the offset parameter. The offset parameter specifies the file position in bytes from the beginning of the file. It returns the error status in EAX.

HLA high-level calling sequence examples:

```
fileio.seek( fileHandle, qwordOffsetVar );
```

HLA low-level calling sequence examples:

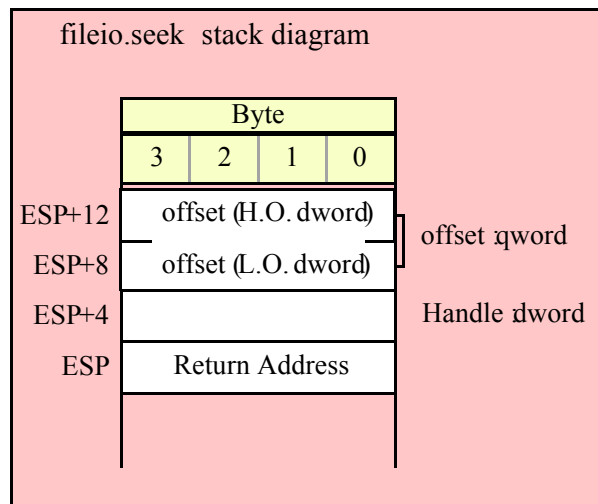
```
push( fileHandle );
push( (type dword qwordOffsetVar[4]));
push( (type dword qwordOffsetVar));
call fileio.seek;
```

```
// If the file handle is in a register (EAX):
```

```
push( eax );
push( (type dword qwordOffsetVar[4]));
push( (type dword qwordOffsetVar));
call fileio.seek;
```

```
// If the offset is in a register pair (EDX:EAX):
```

```
push( fileHandle );
push( edx );    // H.O. dword of offset
push( eax );    // L.O. dword of offset
call fileio.seek;
```



```
fileio.rSeek( Handle:dword; offset:qword ); @returns( "eax" );
```

This function sets the file position in the file specified by the Handle parameter to the position specified by the offset parameter. The offset parameter specifies the file position in bytes from the end of the file. It returns the error status in EAX.

HLA high-level calling sequence examples:

```
fileio.rSeek( fileHandle, qwordOffsetVar );
```

HLA low-level calling sequence examples:

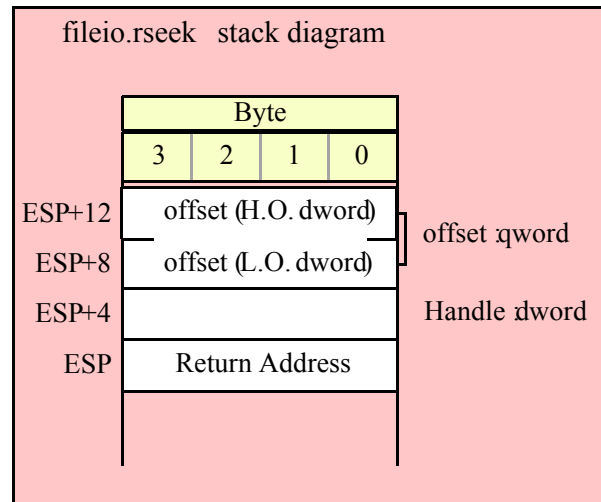
```
push( fileHandle );
push( (type dword qwordOffsetVar[4]));
push( (type dword qwordOffsetVar));
call fileio.rSeek;
```

```
// If the file handle is in a register (EAX):
```

```
push( eax );
push( (type dword qwordOffsetVar[4]));
push( (type dword qwordOffsetVar));
call fileio.rSeek;
```

```
// If the offset is in a register pair (EDX:EAX):
```

```
push( fileHandle );
push( edx );    // H.O. dword of offset
push( eax );    // L.O. dword of offset
call fileio.rSeek;
```



```
fileio.truncate( Handle:dword ); @returns( "eax" );
```

This function deletes all bytes in the file specified by the Handle parameter from the current file position to the end of the file. It returns the error status in EAX.

HLA high-level calling sequence examples:

```
fileio.truncate( fileHandle );
```

```
// If the file handle is in a register (EAX):
```

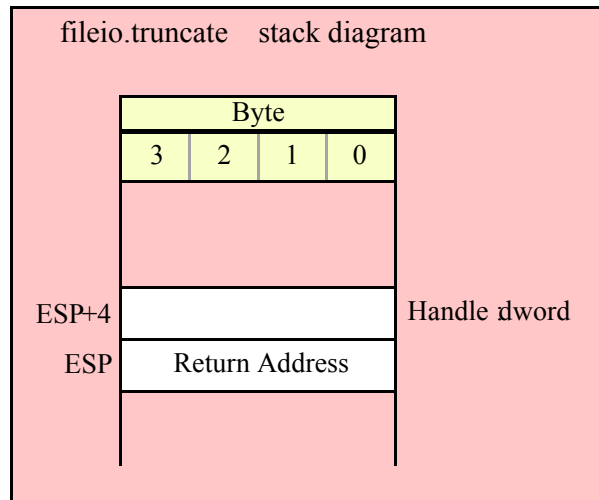
```
fileio.truncate( eax );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
call fileio.truncate;
```

```
// If the file handle is in a register (EAX):
```

```
push( eax );
call fileio.truncate;
```



```
fileio.size( Handle:dword ); @returns( "eax" );
```

This function returns the current size of an open file whose handle you pass as a parameter. It returns the size in the EAX register. Note the overloaded version below.

HLA high-level calling sequence examples:

```
fileio.size( fileHandle );
mov( eax, fileSize );
```

// If the file handle is in a register (EAX):

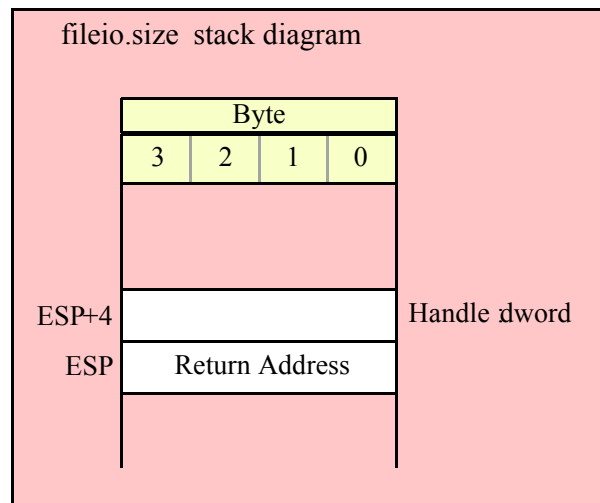
```
fileio.size( eax );
mov( eax, fileSize );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
call fileio.size;
mov( eax, fileSize );
```

// If the file handle is in a register (EAX):

```
push( eax );
call fileio.size;
mov( eax, fileSize );
```



## 15.3 File Output Routines

The file output routines in the fileio module are very similar to the file output routines in the file class module as well as the output routines in the standard output library module. In general, these routines require (at least) two parameters; the first is the file handle that you obtain via the fileio.open or fileio.openNew call, the second parameter is usually the value to write to the file. Some function contain additional parameters that provide formatting information. Note that these functions require that you've opened the file for writing, reading and writing, for for appending. If the file is not open or you've only opened it for reading, these routines will raise an appropriate exception.

### 15.3.1 Miscellaneous Output Routines

```
fileio.write( Handle:dword; var buffer:var; count:uns32 );
```

This procedure writes the number of bytes specified by the count variable to the file. The bytes starting at the address of the buffer byte are written to the file. No range checking is done on the buffer, it is your responsibility to ensure that the buffer contains at least count valid data bytes. Note that buffer is an untyped reference parameter. This means that fileio.write will take the address of whatever object you provide as this parameter (including pointer variables, which may not be what you want). If you want to pass the value of a pointer variable as the buffer address (rather than the address of the pointer variable) when using the high-level style calling syntax, use the VAL keyword as a prefix to the parameter (see the following examples).

HLA high-level calling sequence examples:

```
fileio.write( fileHandle, buffer, count );

// If bufPtr is a dword object containing the
// address of the buffer whose data you wish to
// write to the file:

fileio.write( fileHandle, val bufPtr, count );

// The following writes the four-byte value of
// the bufPtr variable to the file (an unusual
// operation):

fileio.write( fileHandle, bufPtr, 4 );
```

HLA low-level calling sequence examples:

```

// Assumes buffer is a static object at a fixed
// address in memory:

push( fileHandle );
pushd( &buffer );
push( count );
call fileio.write;

    // If a 32-bit register is available and buffer
    // isn't at a fixed, static, address:

push( fileHandle );
lea( eax, buffer );
push( eax );
push( count );
call fileio.write;

    // If a 32-bit register is not available and buffer
    // isn't at a fixed, static, address:

push( fileHandle );
sub( 4, esp );
push( eax );
lea( eax, buffer );
mov( eax, [esp+4] );
pop( eax );
push( count );
call fileio.write;

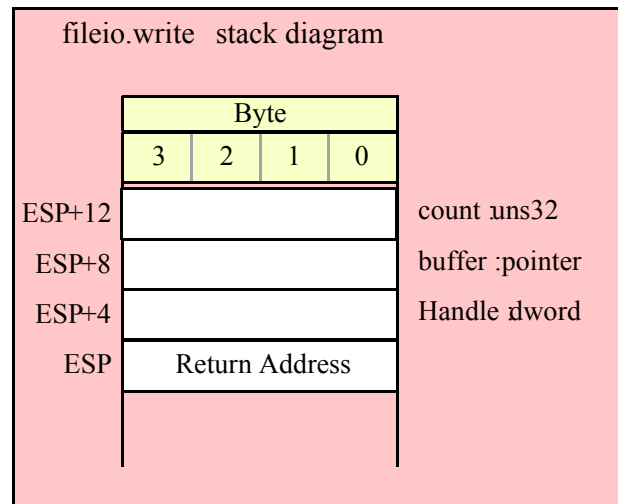
// If bufPtr points at the buffer to write,
// then use code like this:

push( fileHandle );
push( bufPtr );
push( count );
call fileio.write;

    // To write the 4 bytes at bufPtr to
    // the file (unusual), you could use
    // code like this:

push( fileHandle );
lea( eax, bufPtr );
push( eax );
pushd( 4 );
call fileio.write;

```



```
fileio.newln( Handle:dword )
```

This function writes a newline sequence (e.g., carriage return/line feed under Windows or line feed under Linux) to the specified output file.

HLA high-level calling sequence examples:

```
fileio.newln( fileHandle );
```

```
// If the file handle is in a register (EAX):
```

```
fileio.newln( eax );
```

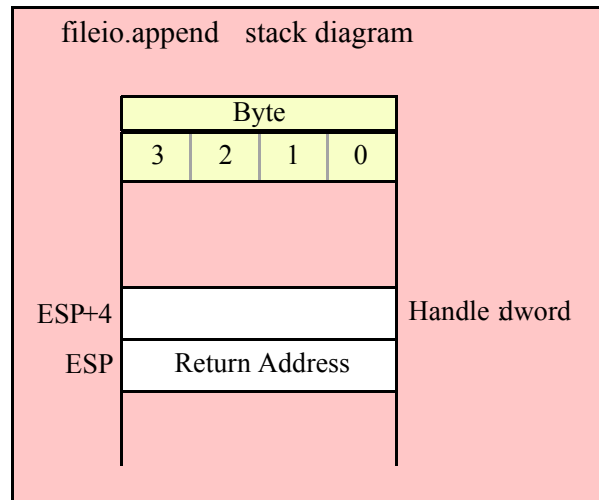
HLA low-level calling sequence examples:

```
push( fileHandle );
call fileio.newln;
```

```
// If the file handle is in a register (EAX):
```

```
push( eax );
call fileio.newln;
```





**fileio.putbool( Handle:dword; b:boolean )**

This procedure writes the string "true" or "false" to the output file depending on the value of the b parameter.

HLA high-level calling sequence examples:

```
fileio.putbool( fileHandle, boolVar );
```

```
// If the boolean is in a register (AL):
```

```
fileio.putbool( fileHandle, al );
```

HLA low-level calling sequence examples:

```
// If "boolVar" is not one of the last three
// bytes on a page of memory, you can do this:
```

```
push( fileHandle );
push( (type dword boolVar) );
call fileio.putbool;
```

```
// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:
```

```
push( fileHandle );
movzx( boolVar, eax ); // Assume EAX is available
push( eax );
call fileio.putbool;
```

```
// If no register is available, do something
// like the following code:
```

```
push( fileHandle );
sub( 4, esp );
push( eax );
```

```

movzx( boolVar , eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.putbool;

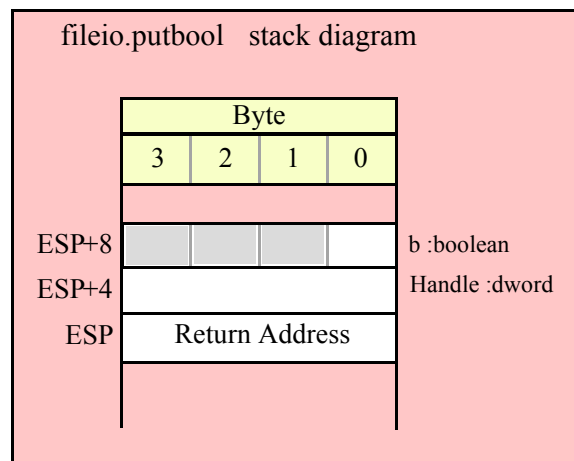
// If the boolean value is in al, bl, cl, or dl
// then you can use code like the following:

push( fileHandle );
push( eax ); // Assume boolVar is in AL
call fileio.putbool;

// If the Boolean value is in ah, bh, ch, or dh
// you'll have to use code like the following:

push( fileHandle );
xchg( al, ah ); // Assume boolVar is in AH
push( eax );    // It's now in AL
xchg( al, ah ); // Restore al/ah
call fileio.putbool;

```



## 15.3.2 Character, String, and Character Set Output Routines

**fileio.putc( Handle:dword; c:char )**

Writes the character specified by the `c` parameter to the file specified by the `Handle` parameter.

HLA high-level calling sequence examples:

```
fileio.putc( fileHandle, charVar );
```

```
// If the character is in a register (AL):
```

```
fileio.putc( fileHandle, al );
```

HLA low-level calling sequence examples:

```
// If "charVar" is not one of the last three
// bytes on a page of memory, you can do this:

push( fileHandle );
push( (type dword charVar) );
call fileio.putc;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

push( fileHandle );
movzx( charVar, eax ); // Assume EAX is available
push( eax );
call fileio.putc;

// If no register is available, do something
// like the following code:

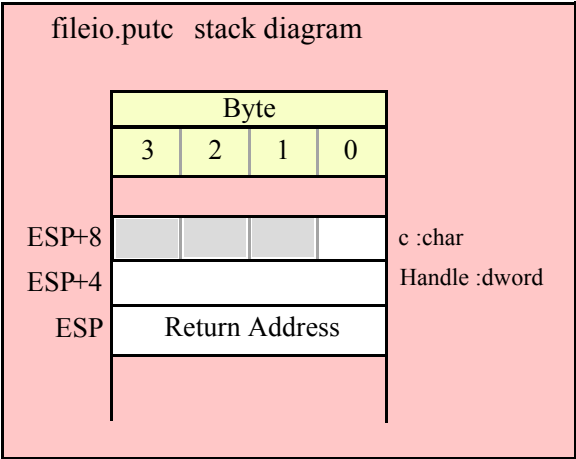
push( fileHandle );
sub( 4, esp );
push( eax );
movzx( charVar, eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.putc;

// If the character value is in al, bl, cl, or dl
// then you can use code like the following:

push( fileHandle );
push( eax ); // Assume charVar is in AL
call fileio.putc;

// If the character value is in ah, bh, ch, or dh
// you'll have to use code like the following:

push( fileHandle );
xchg( al, ah ); // Assume charVar is in AH
push( eax );    // It's now in AL
xchg( al, ah ); // Restore al/ah
call fileio.putc;
```



**fileio.putcSize( Handle:dword; c:char; width:int32; fill:char )**

Outputs the character *c* to the file using at least *width* output positions. If the absolute value of *width* is greater than one, then this function writes *fill* characters as padding characters during the output. If *width* is a positive value greater than one, then `fileio.putcSize` writes *c* left justified in a field of *width* characters; if *width* is a negative value less than one, then `fileio.putcSize` writes *c* right justified in a field of *width* characters.

HLA high-level calling sequence examples:

```
fileio.putcSize( fileHandle, charVar, width, padChar );
```

HLA low-level calling sequence examples:

```
// If "charVar" and "padChar" are not one of the last three
// bytes on a page of memory, you can do this:
```

```
push( fileHandle );
push( (type dword charVar) );
push( width );
push( (type dword padChar) );
call fileio.putcSize;
```

```
// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:
```

```
push( fileHandle );
movzx( charVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call fileio.putcSize;
```

```
// If no registers are available, do something
// like the following code:
```

```
push( fileHandle );
sub( 12, esp );
push( eax );
movzx( charVar, eax );
mov( eax, [esp+12] );
mov( width, eax );
mov( eax, [esp+8] );
movzx( padChar, eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.putcSize;
```

```
// If "charVar" or "padChar" are in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:
```

```
push( fileHandle );
push( eax ); // Assume charVar is in AL
```

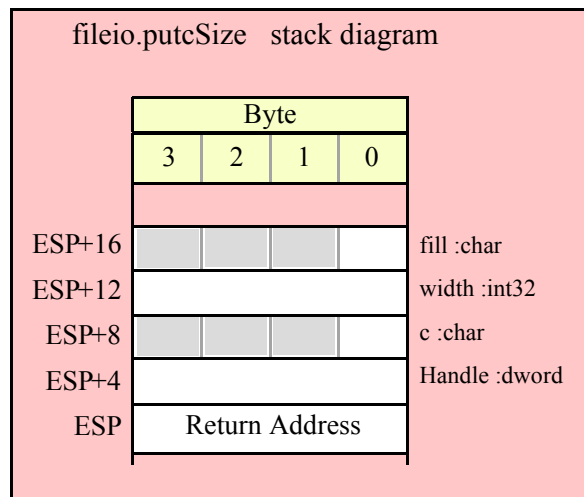
```

push( width );
push( ebx );    // Assume padChar is in BL
call fileio.putcSize;

// Do the following if the characters are
// in AH, BH, CH, or DH:

push( fileHandle );
xchg( al, ah );    // Assume charVar is in AH
xchg( bl, bh );    // Assume padChar is in BH
push( eax );
push( width );
push( ebx );
xchg( al, ah );
xchg( bl, bh );
call fileio.putcSize;

```



```
fileio.putcset( Handle:dword; cst:cset )
```

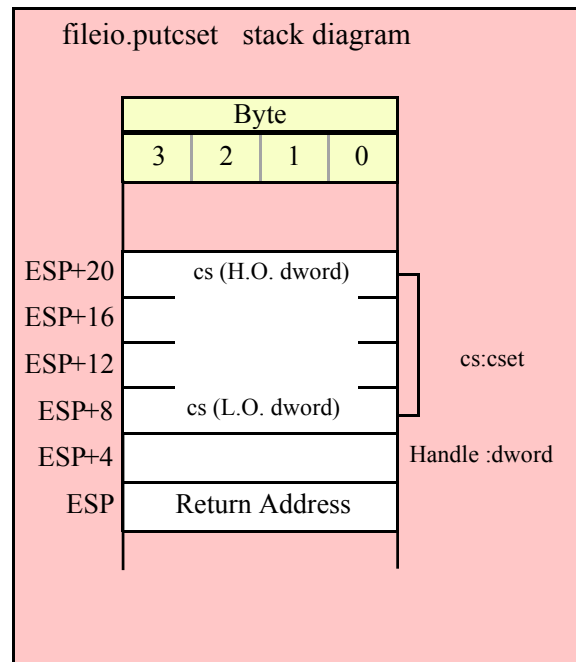
This function writes all the members of the `cst` character set parameter to the file specified by the `Handle` variable.

HLA high-level calling sequence examples:

```
fileio.putcset( fileHandle, csVar );
fileio.putcset( fileHandle, [ebx] ); // EBX points at the cset.
```

HLA low-level calling sequence examples:

```
push( fileHandle );
push( (type dword csVar[12]) ); // Push H.O. dword first
push( (type dword csVar[8]) );
push( (type dword csVar[4]) );
push( (type dword csVar) );      // Push L.O. dword last
call fileio.putcset;
```



```
fileio.puts( Handle:dword; s:string )
```

This procedure writes the value of the string parameter to the specified file. Remember, string values are actually 4-byte pointers to the string's character data.

HLA high-level calling sequence examples:

```
fileio.puts( fileHandle, strVar );
fileio.puts( fileHandle, ebx ); // EBX holds a string value.
fileio.puts( fileHandle, "Hello World" );
```

HLA low-level calling sequence examples:

```
// For string variables:
```

```
push( fileHandle );
push( strVar );
call fileio.puts;
```

```
// For string values held in registers:
```

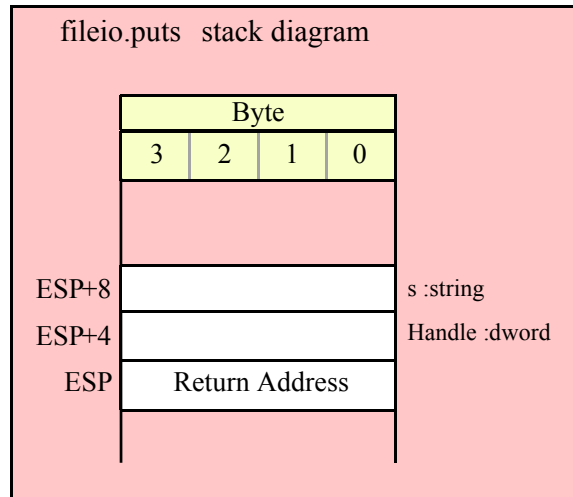
```
push( fileHandle );
push( ebx ); // Assume EBX holds the string value
call fileio.puts;
```

```
// For string literals, assuming a 32-bit register
// is available:
```

```
push( fileHandle );
lea( eax, "Hello World" ); // Assume EAX is available.
push( eax );
call fileio.puts;
```

```
// If a 32-bit register is not available:

readonly
  literalString :string := "Hello World";
  .
  .
  .
push( fileHandle );
push( literalString );
call fileio.puts;
```



**fileio.putsSize( Handle:dword; s:string; width:int32; fill:char )**

This function writes the s string to the file using at least width character positions. If the absolute value of width is less than or equal to the length of s, then this function behaves exactly like fileio.puts. On the other hand, if the absolute value of width is greater than the length of s, then fileio.putsSize writes width characters to the output file. This procedure emits the fill character in the extra print positions. If width is positive, then fileio.putsSize right justifies the string in the print field. If width is negative, then fileio.putsSize left justifies the string in the print field. Generally, people expect the string to be left justified, so you should ensure that this value is negative to achieve this.

HLA high-level calling sequence examples:

```
fileio.putsSize( fileHandle, strVar, width, ' ' );

// For the following, EBX holds the string value,
// ECX contains the width, and AL holds the pad
// character:

fileio.putsSize( fileHandle, ebx, ecx, al );

fileio.putsSize( fileHandle, "Hello World", 25, padChar );
```

HLA low-level calling sequence examples:

```
// For string variables:

push( fileHandle );
push( strVar );
```

```

push( width );
pushd( ' ' );
call fileio.putsSize;

// For string values held in registers:

push( fileHandle );
push( ebx ); // Assume EBX holds the string value
push( ecx ); // Assume ECX holds the width
push( eax ); // Assume AL holds the fill character
call fileio.putsSize;

// For string literals, assuming a 32-bit register
// is available:

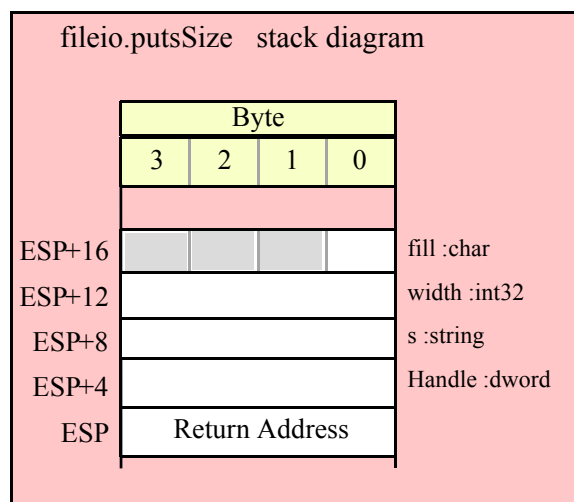
push( fileHandle );
lea( eax, "Hello World" ); // Assume EAX is available.
push( eax );
pushd( 25 );
movzx( padChar, eax );
push( eax );
call fileio.putsSize;

// If a 32-bit register is not available:

readonly
    literalString :string := "Hello World";

    // Note: element zero is the actual pad character.
    // The other elements are just padding.
    padChar :char[4] := [ '.', #0, #0, #0 ];
    .
    .
    .
push( fileHandle );
push( literalString );
pushd( 25 );
push( (type dword padChar) );
call fileio.putsSize;

```





### 15.3.3 Hexadecimal Output Routines

**fileio.putb( Handle:dword; b:byte )**

This procedure writes the value of b to the file using exactly two hexadecimal digits (including a leading zero if necessary).

HLA high-level calling sequence examples:

```
fileio.putb( fileHandle, byteVar );

// If the character is in a register (AL):

fileio.putb( fileHandle, al );
```

HLA low-level calling sequence examples:

```
// If "byteVar " is not one of the last three
// bytes on a page of memory, you can do this:

push( fileHandle );
push( (type dword byteVar ) );
call fileio.putb;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

push( fileHandle );
movzx( byteVar , eax ); // Assume EAX is available
push( eax );
call fileio.putb;

// If no register is available, do something
// like the following code:

push( fileHandle );
sub( 4, esp );
push( eax );
movzx( byteVar , eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.putb;

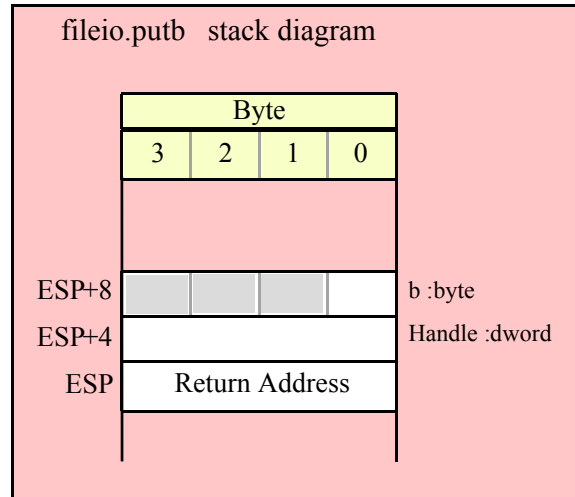
// If the character value is in al, bl, cl, or dl
// then you can use code like the following:

push( fileHandle );
push( eax ); // Assume byteVar is in AL
call fileio.putb;

// If the byte value is in ah, bh, ch, or dh
// you'll have to use code like the following:

push( fileHandle );
xchg( al, ah ); // Assume byteVar is in AH
push( eax );    // It's now in AL
```

```
xchg( al, ah ); // Restore al/ah
call fileio.putb;
```



```
fileio.puth8( Handle:dword; b:byte )
```

This procedure writes the value of b to the file using the minimum necessary number of hexadecimal digits.

HLA high-level calling sequence examples:

```
fileio.puth8( fileHandle, byteVar );
```

```
// If the character is in a register (AL):
```

```
fileio.puth8( fileHandle, al );
```

HLA low-level calling sequence examples:

```
// If "byteVar " is not one of the last three
// bytes on a page of memory, you can do this:
```

```
push( fileHandle );
push( (type dword byteVar ) );
call fileio.puth8;
```

```
// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:
```

```
push( fileHandle );
movzx( byteVar , eax ); // Assume EAX is available
push( eax );
call fileio.puth8;
```

```
// If no register is available, do something
// like the following code:
```

```
push( fileHandle );
```

```

sub( 4, esp );
push( eax );
movzx( byteVar , eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.puth8;

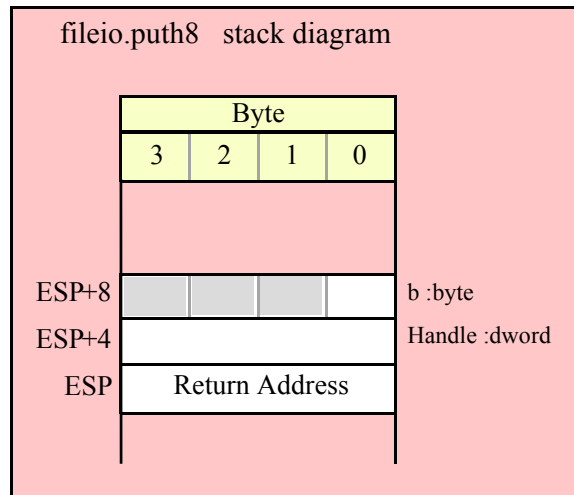
// If the character value is in al, bl, cl, or dl
// then you can use code like the following:

push( fileHandle );
push( eax ); // Assume byteVar is in AL
call fileio.puth8;

// If the byte value is in ah, bh, ch, or dh
// you'll have to use code like the following:

push( fileHandle );
xchg( al, ah ); // Assume byteVar is in AH
push( eax );    // It's now in AL
xchg( al, ah ); // Restore al/ah
call fileio.puth8;

```



**fileio.puth8Size( Handle:dword; b:byte; size:dword; fill:char )**

The fileio.puth8Size function writes an 8-bit hexadecimal value to a file allowing you specify a minimum field width and a fill character.

HLA high-level calling sequence examples:

```
fileio.puth8Size( fileHandle, byteVar, width, padChar );
```

HLA low-level calling sequence examples:

```

// If "byteVar" and "padChar" are not one of the last three
// bytes on a page of memory, you can do this:

```

```

push( fileHandle );
push( (type dword byteVar) );
push( width );
push( (type dword padChar) );
call fileio.puth8Size;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

push( fileHandle );
movzx( byteVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call fileio.puth8Size;

// If no registers are available, do something
// like the following code:

push( fileHandle );
sub( 12, esp );
push( eax );
movzx( byteVar, eax );
mov( eax, [esp+12] );
mov( width, eax );
mov( eax, [esp+8] );
movzx( padChar, eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.puth8Size;

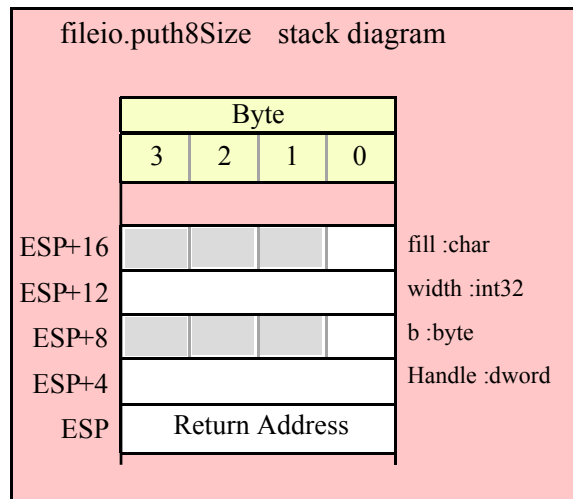
// If "byteVar" or "padChar" are in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:

push( fileHandle );
push( eax ); // Assume byteVar is in AL
push( width );
push( ebx ); // Assume padChar is in BL
call fileio.puth8Size;

// Do the following if the characters are
// in AH, BH, CH, or DH:

push( fileHandle );
xchg( al, ah ); // Assume byteVar is in AH
xchg( bl, bh ); // Assume padChar is in BH
push( eax );
push( width );
push( ebx );
xchg( al, ah );
xchg( bl, bh );
call fileio.puth8Size;

```



**fileio.putw( Handle:dword; w:word )**

This procedure writes the value of w to the file using exactly four hexadecimal digits (including leading zeros if necessary).

HLA high-level calling sequence examples:

```
fileio.putw( fileHandle, wordVar );

// If the word is in a register (AX):

fileio.putw( fileHandle, ax );
```

HLA low-level calling sequence examples:

```
// If "wordVar " is not one of the last three
// bytes on a page of memory, you can do this:

push( fileHandle );
push( (type dword wordVar) );
call fileio.putw;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

push( fileHandle );
movzx( wordVar, eax ); // Assume EAX is available
push( eax );
call fileio.putw;

// If no register is available, do something
// like the following code:

push( fileHandle );
```

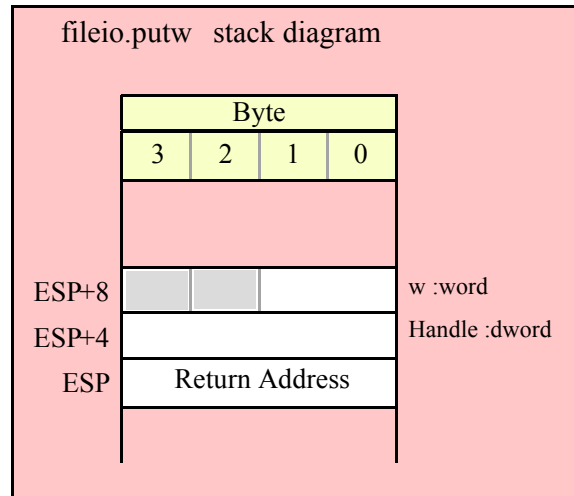
```

sub( 4, esp );
push( eax );
movzx( wordVar, eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.putw;

// If the word value is in a 16-bit register
// then you can use code like the following:

push( fileHandle );
push( eax ); // Assume wordVar is in AX
call fileio.putw;

```



**fileio.puth16( Handle:dword; w:word )**

This procedure writes the value of `w` to the file using the minimum necessary number of hexadecimal digits.

HLA high-level calling sequence examples:

```
fileio.puth16( fileHandle, wordVar );
```

```
// If the word is in a register (AX):
```

```
fileio.puth16( fileHandle, ax );
```

HLA low-level calling sequence examples:

```
// If "wordVar " is not one of the last three
// bytes on a page of memory, you can do this:
```

```
push( fileHandle );
push( (type dword wordVar) );
call fileio.puth16;
```

```
// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
```

```

// the following:

push( fileHandle );
movzx( wordVar, eax ); // Assume EAX is available
push( eax );
call fileio.puth16;

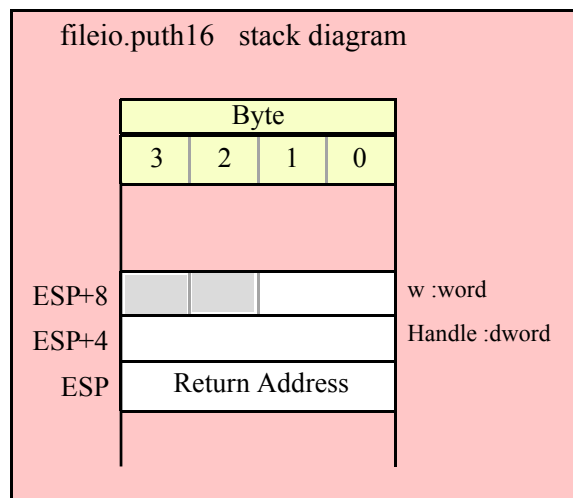
// If no register is available, do something
// like the following code:

push( fileHandle );
sub( 4, esp );
push( eax );
movzx( wordVar, eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.puth16;

// If the word value is in a 16-bit register
// then you can use code like the following:

push( fileHandle );
push( eax ); // Assume wordVar is in AX
call fileio.puth16;

```



**fileio.puth16Size( Handle:dword; w:word; size:dword; fill:char )**

The `fileio.puth16Size` function writes a 16-bit hexadecimal value to a file allowing you specify a minimum field width and a fill character.

HLA high-level calling sequence examples:

```
fileio.puth16Size( fileHandle, wordVar, width, padChar );
```

HLA low-level calling sequence examples:

```

// If "wordVar" and "padChar" are not one of the last three
// bytes on a page of memory, you can do this:

```

```

push( fileHandle );
push( (type dword wordVar) );
push( width );
push( (type dword padChar) );
call fileio.puth16Size;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

push( fileHandle );
movzx( wordVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call fileio.puth16Size;

// If no registers are available, do something
// like the following code:

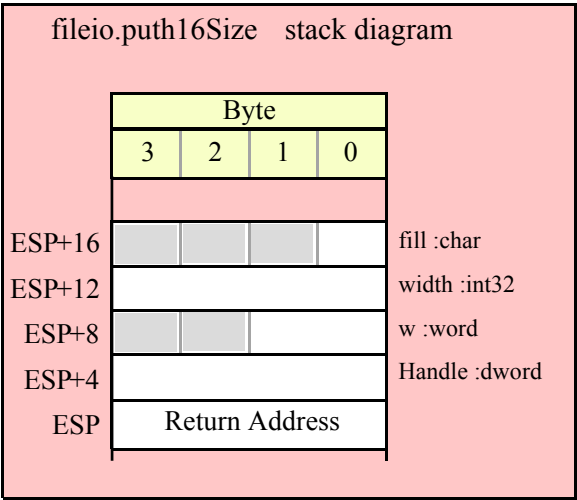
push( fileHandle );
sub( 12, esp );
push( eax );
movzx( wordVar, eax );
mov( eax, [esp+12] );
mov( width, eax );
mov( eax, [esp+8] );
movzx( padChar, eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.puth16Size;

// If "wordVar" is in a 16-bit register
// and "padChar" is in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:

push( fileHandle );
push( eax ); // Assume wordVar is in AX
push( width );
push( ebx ); // Assume padChar is in BL
call fileio.puth16Size;

```





**fileio.putd( Handle:dword; d:dword )**

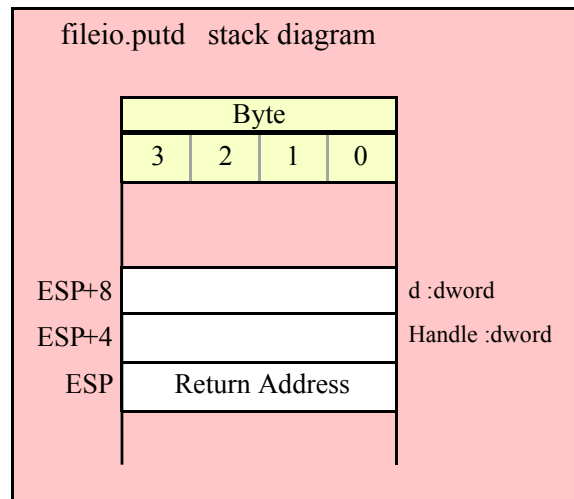
This procedure writes the value of d to the file using exactly eight hexadecimal digits (including leading zeros if necessary), if underscore output is not enabled. This routine will emit nine characters (eight digits plus an underscore) if underscore output is enabled.

HLA high-level calling sequence examples:

```
fileio.putd( fileHandle, dwordVar );  
  
// If the dword value is in a register (EAX):  
  
fileio.putd( fileHandle, eax );
```

HLA low-level calling sequence examples:

```
push( fileHandle );  
push( dwordVar );  
call fileio.putd;  
  
push( fileHandle );  
push( eax );  
call fileio.putd;
```



**fileio.puth32( Handle:dword; d:dword )**

This procedure writes the value of d to the file using the minimum number of hexadecimal digits necessary. If underscore output is enabled (see conv.setUnderscores and conv.getUnderscores) then this function will emit an underscore between groups of four hexadecimal digits, starting from the least significant digit.

HLA high-level calling sequence examples:

```
fileio.puth32( fileHandle, dwordVar );
```

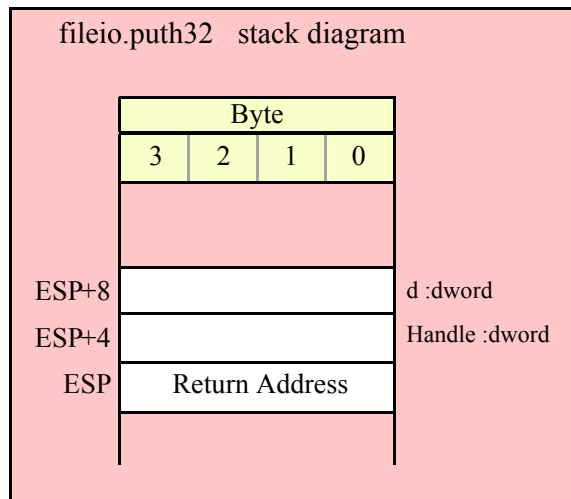
```
// If the dword is in a register (EAX):
```

```
fileio.puth32( fileHandle, eax );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
push( dwordVar );
call fileio.puth32;
```

```
push( fileHandle );
push( eax );
call fileio.puth32;
```



```
fileio.puth32Size( Handle:dword; d:dword; size:dword; fill:char )
```

The `fileio.path32Size` function outputs `d` as a hexadecimal string (including underscores, if enabled) and it allows you specify a minimum field width and a fill character.

HLA high-level calling sequence examples:

```
fileio.puth32Size( fileHandle, dwordVar, width, ' ' );
```

```
// If the dword is in a register (EAX):
```

```
fileio.puth32Size( fileHandle, eax, width, cl );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
push( dwordVar );
push( width );
pushd( ' ' );
call fileio.puth32Size;
```

```
push( fileHandle );
push( eax );
push( width );
push( ecx ); // fill char is in CL
call fileio.puth32Size;
```

```
// Assume fill char is in CH
```

```
push( fileHandle );
push( eax );
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call fileio.puth32Size;
```

```

// Alternate method of the above

push( fileHandle );
push( eax );
push( width );
sub( 4, esp );
mov( ch, [esp] );
call fileio.puth32Size;

// If the fill char is a variable and
// a register is available, try this code:

push( fileHandle );
push( eax );
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call fileio.puth32Size;

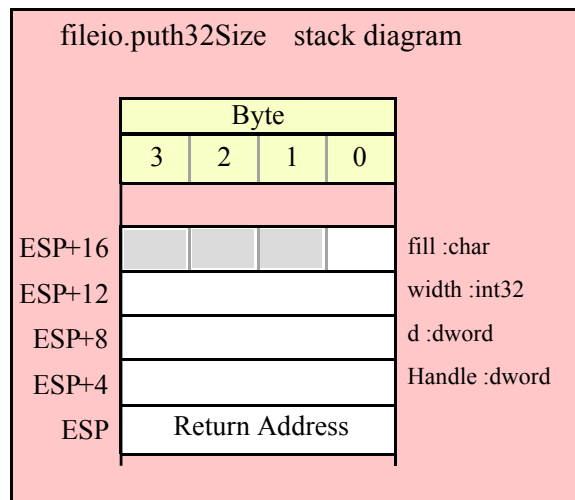
// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( fileHandle );
push( eax );
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call fileio.puth32Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( fileHandle );
push( eax );
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.puth32Size;

```



**fileio.putq( Handle:dword; q:qword )**

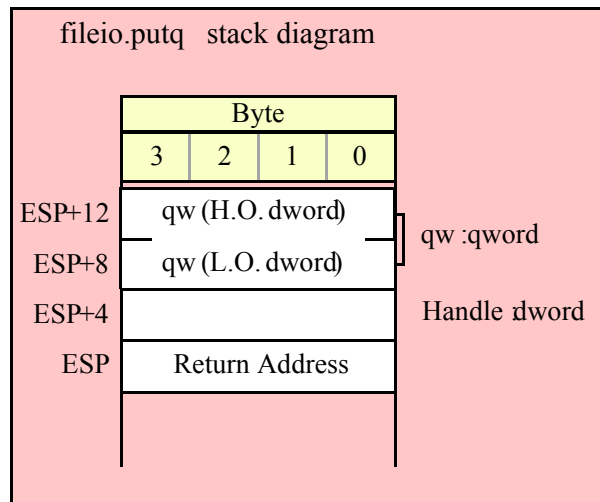
This procedure writes the value of q to the file using exactly sixteen hexadecimal digits (including leading zeros if necessary and an intervening underscores if underscore output is enabled).

HLA high-level calling sequence examples:

```
fileio.putq( fileHandle, qwordVar );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
call fileio.putq;
```



**fileio.puth64( Handle:dword; q:qword )**

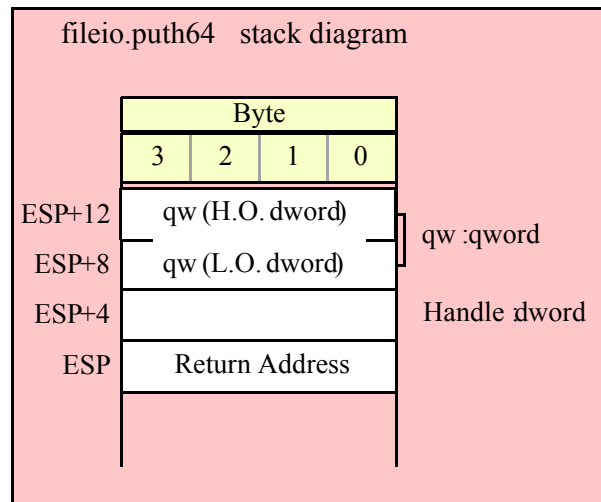
This procedure writes the value of q to the file using the minimum necessary number of hexadecimal digits (including intervening underscores if underscore output is enabled).

HLA high-level calling sequence examples:

```
fileio.puth64( fileHandle, qwordVar );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
call fileio.puth64;
```



**fileio.puth64Size( Handle:dword; q:qword; size:dword; fill:char )**

The fileio.putqSize function lets you specify a minimum field width and a fill character. The fileio.putq routine uses a minimum size of two and a fill character of '0'. Note that if underscore output is enabled, this routine will emit 19 characters (16 digits plus three underscores).

HLA high-level calling sequence examples:

```
fileio.puth64Size( fileHandle, qwordVar, width, ' ' );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
pushd( ' ' );
call fileio.puth64Size;
```

```
push( fileHandle );
push( edx ); // Assume 64-bit value in edx:eax
push( eax );
push( width );
push( ecx ); // fill char is in CL
call fileio.puth64Size;
```

```
// Assume fill char is in CH
```

```
push( fileHandle );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call fileio.puth64Size;
```

```

// Alternate method of the above

push( fileHandle );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
sub( 4, esp );
mov( ch, [esp] );
call fileio.puth64Size;

// If the fill char is a variable and
// a register is available, try this code:

push( fileHandle );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call fileio.puth64Size;

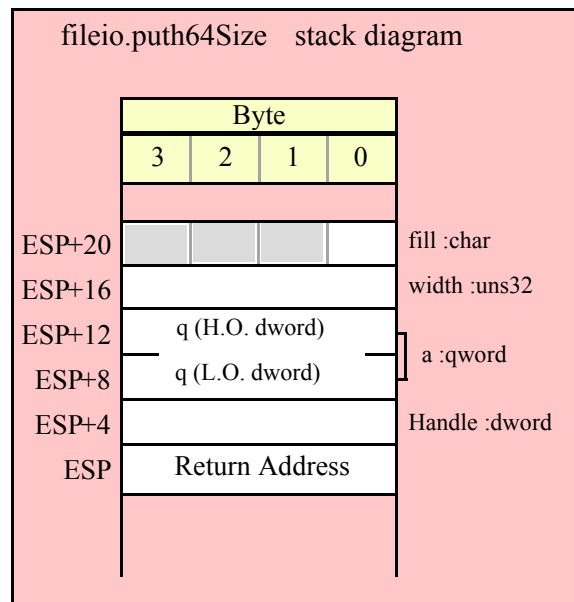
// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( fileHandle );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call fileio.puth64Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( fileHandle );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.puth64Size;

```



```
fileio.puttb( Handle:dword; tb:tbyte )
```

This procedure writes the value of `tb` to the file using exactly 20 hexadecimal digits (including leading zeros if necessary and an intervening underscores if underscore output is enabled).

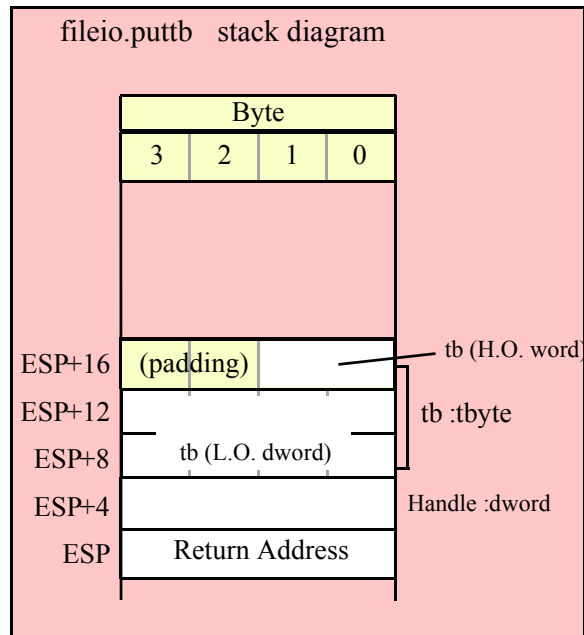
HLA high-level calling sequence examples:

```
fileio.puttb( fileHandle, tbyteVar );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
pushw( 0 ); // Push push a 0 pad word
push( (type word tbyteVar[8])); // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar)); // L.O. dword last
call fileio.puttb;
```





**fileio.puth80( Handle:dword; tb:tbyte )**

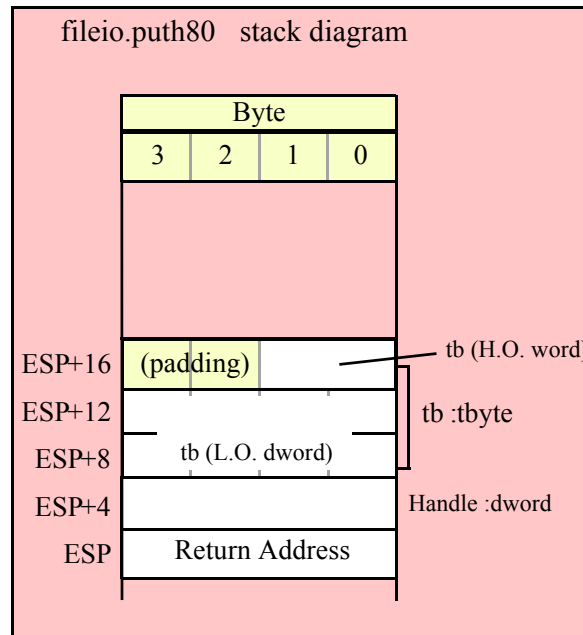
This procedure writes the value of tb to the file using the minimum necessary number of hexadecimal digits (including intervening underscores if underscore output is enabled).

HLA high-level calling sequence examples:

```
fileio.puth80( fileHandle, tbyteVar );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
pushw( 0 );           // Push push a 0 pad word
push( (type word tbyteVar[8])); // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));    // L.O. dword last
call fileio.puth80;
```



**fileio.puth80Size( Handle:dword; tb:tbyte; size:dword; fill:char )**

The fileio.puth80Size function lets you specify a minimum field width and a fill character. It writes the tbyte value tb as a hexadecimal string to the specified file using the provided minimum size and fill character.

HLA high-level calling sequence examples:

```
fileio.puth80Size( fileHandle, tbyteVar, width, ' ' );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
pushw( 0 );           // Push push a 0 pad word
push( (type word tbyteVar[8])); // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));    // L.O. dword last
push( width );
pushd( ' ' );
call fileio.puth80Size;
```

```
// Assume fill char is in CH
```

```
push( fileHandle );
pushw( 0 );           // Push push a 0 pad word
push( (type word tbyteVar[8])); // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));    // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
```

```

call fileio.puth80Size;

// Alternate method of the above

push( fileHandle );
pushw( 0 );           // Push push a 0 pad word
push( (type word tbyteVar[8])); // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));    // L.O. dword last
push( width );
sub( 4, esp );
mov( ch, [esp] );
call fileio.puth80Size;

// If the fill char is a variable and
// a register is available, try this code:

push( fileHandle );
pushw( 0 );           // Push push a 0 pad word
push( (type word tbyteVar[8])); // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));    // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call fileio.puth80Size;

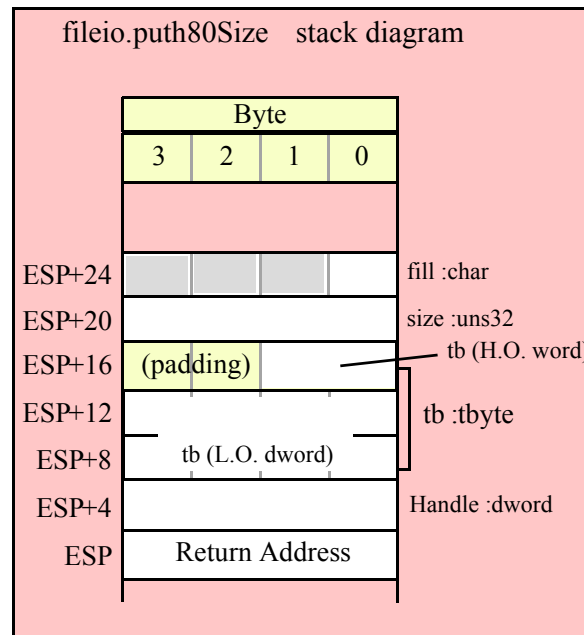
// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( fileHandle );
pushw( 0 );           // Push push a 0 pad word
push( (type word tbyteVar[8])); // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));    // L.O. dword last
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call fileio.puth80Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( fileHandle );
pushw( 0 );           // Push push a 0 pad word
push( (type word tbyteVar[8])); // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));    // L.O. dword last
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.puth80Size;

```



```
fileio.putl( Handle:dword; l:lword )
```

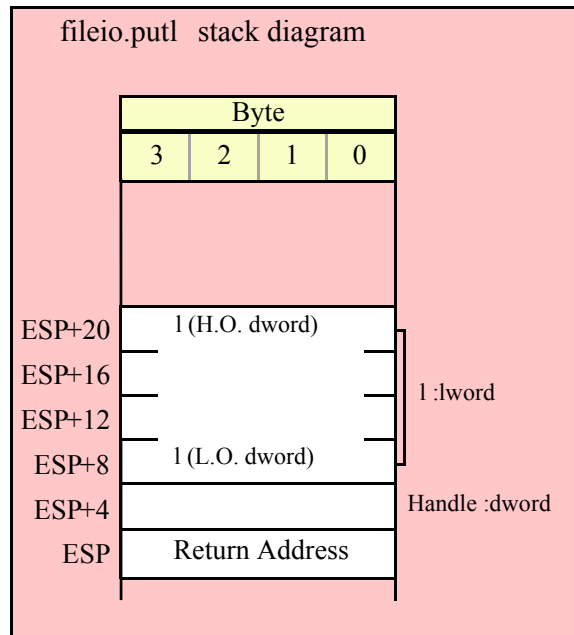
This procedure writes the value of `l` to the file using exactly 32 hexadecimal digits (including leading zeros if necessary and an intervening underscores if underscore output is enabled).

HLA high-level calling sequence examples:

```
fileio.putl( fileHandle, lwordVar );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
push( (type dword lwordVar[12]) ); // H.O. dword first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
call fileio.putl;
```



**fileio.puth128( Handle:dword; l:lword )**

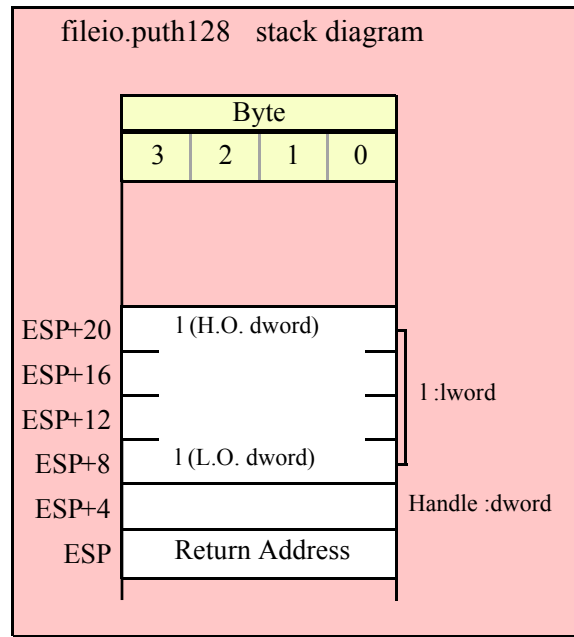
This procedure writes the value of *l* to the file using the minimum necessary number of hexadecimal digits (including intervening underscores if underscore output is enabled).

HLA high-level calling sequence examples:

```
fileio.puth128( fileHandle, lwordVar );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
push( (type dword lwordVar[12]) ); // H.O. dword first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
call fileio.puth128;
```



**fileio.puth128Size( Handle:dword; 1:1word; size:dword; fill:char )**

The fileio.puth128Size function writes an lword value to the file and it lets you specify a minimum field width and a fill character.

HLA high-level calling sequence examples:

```
fileio.puth128Size( fileHandle, tbyteVar, width, ' ' );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));    // L.O. dword last
push( width );
pushd( ' ' );
call fileio.puth128Size;
```

```
// Assume fill char is in CH
```

```
push( fileHandle );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));    // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
```

```

call fileio.puth128Size;

// Alternate method of the above

push( fileHandle );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));    // L.O. dword last
push( width );
sub( 4, esp );
mov( ch, [esp] );
call fileio.puth128Size;

// If the fill char is a variable and
// a register is available, try this code:

push( fileHandle );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));    // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call fileio.puth128Size;

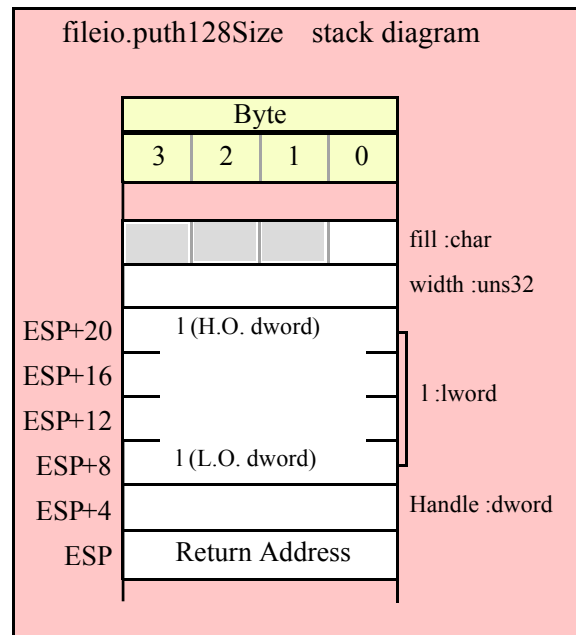
// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( fileHandle );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));    // L.O. dword last
push( width );
push( (type dword fillChar) );    // Chance of page crossing!
call fileio.puth128Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( fileHandle );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));    // L.O. dword last
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.puth128Size;

```



### 15.3.4 Signed Integer Output Routines

These routines convert signed integer values to string format and write that string to the file specified by the Handle parameter. The fileio.putxxxSize functions contain width and fill parameters that let you specify the minimum field width when outputting a value.

If the absolute value of width is greater than the number of print positions the value requires, then these functions output width characters to the output file. If width is non-negative, then these functions right-justify the value in the output field; if value is negative, then these functions left-justify the value in the output field.

These functions print the fill character as the padding value for the extra print positions.

Note that unlike floating point values, these functions do not print a space in front of the value if it is non-negative.

```
xxxSize( value, width, fill );
```

Assuming “value” requires five print positions, “width” is eight, and fill is “f” then the xxxSize functions produce the string

f	f	f	V	A	L	U	E
---	---	---	---	---	---	---	---

Assuming “value” requires five print positions, “width” is minus eight, and fill is “f” then the xxxSize functions produce the string

V	A	L	U	E	f	f	f
---	---	---	---	---	---	---	---

```
fileio.puti8 ( Handle:dword; b:byte )
```

This function converts the eight-bit signed integer you pass as a parameter to a string and writes this string to the file (specified by Handle) using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
fileio.puti8( fileHandle, byteVar );
```



```
// If the character is in a register (AL):
```

```
fileio.puti8( fileHandle, al );
```

HLA low-level calling sequence examples:

```
// If "byteVar " is not one of the last three
// bytes on a page of memory, you can do this:
```

```
push( fileHandle );
push( (type dword byteVar ) );
call fileio.puti8;
```

```
// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:
```

```
push( fileHandle );
movzx( byteVar , eax ); // Assume EAX is available
push( eax );
call fileio.puti8;
```

```
// If no register is available, do something
// like the following code:
```

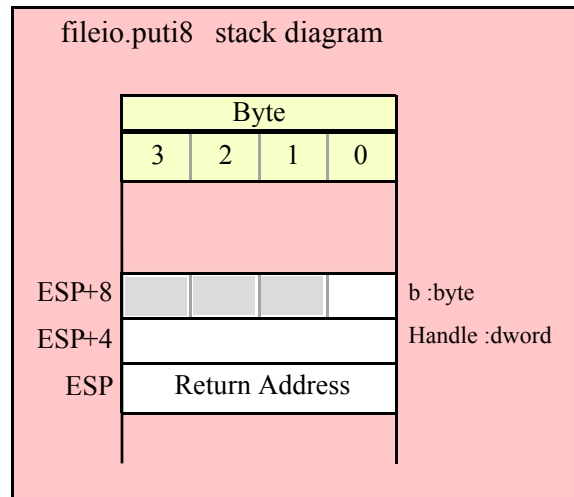
```
push( fileHandle );
sub( 4, esp );
push( eax );
movzx( byteVar , eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.puti8;
```

```
// If the character value is in al, bl, cl, or dl
// then you can use code like the following:
```

```
push( fileHandle );
push( eax ); // Assume byteVar is in AL
call fileio.puti8;
```

```
// If the byte value is in ah, bh, ch, or dh
// you'll have to use code like the following:
```

```
push( fileHandle );
xchg( al, ah ); // Assume byteVar is in AH
push( eax );    // It's now in AL
xchg( al, ah ); // Restore al/ah
call fileio.puti8;
```



**fileio.puti8Size ( Handle:dword; b:byte; width:int32; fill:char )**

This function writes the eight-bit signed integer value you pass to the specified output file using the width and fill values as specified above.

HLA high-level calling sequence examples:

```
fileio.puti8Size( fileHandle, byteVar, width, padChar );
```

HLA low-level calling sequence examples:

```
// If "byteVar" and "padChar" are not one of the last three
// bytes on a page of memory, you can do this:

push( fileHandle );
push( (type dword byteVar) );
push( width );
push( (type dword padChar) );
call fileio.puti8Size;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

push( fileHandle );
movzx( byteVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call fileio.puti8Size;

// If no registers are available, do something
// like the following code:

push( fileHandle );
sub( 12, esp );
push( eax );
```

```

movzx( byteVar, eax );
mov( eax, [esp+12] );
mov( width, eax );
mov( eax, [esp+8] );
movzx( padChar, eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.puti8Size;

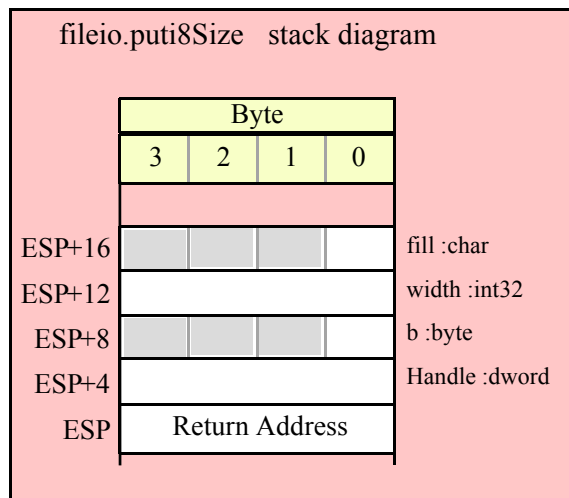
// If "byteVar" or "padChar" are in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:

push( fileHandle );
push( eax );    // Assume byteVar is in AL
push( width );
push( ebx );    // Assume padChar is in BL
call fileio.puti8Size;

// Do the following if the characters are
// in AH, BH, CH, or DH:

push( fileHandle );
xchg( al, ah );    // Assume byteVar is in AH
xchg( bl, bh );    // Assume padChar is in BH
push( eax );
push( width );
push( ebx );
xchg( al, ah );
xchg( bl, bh );
call fileio.puti8Size;

```



**fileio.puti16( Handle:dword; w:word )**

This function converts the 16-bit signed integer you pass as a parameter to a string and writes this string to the file (specified by Handle) using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```

fileio.puti16( fileHandle, wordVar );

// If the word is in a register (AX):

fileio.puti16( fileHandle, ax );

```

HLA low-level calling sequence examples:

```

// If "wordVar " is not one of the last three
// bytes on a page of memory, you can do this:

push( fileHandle );
push( (type dword wordVar) );
call fileio.puti16;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

push( fileHandle );
movzx( wordVar, eax ); // Assume EAX is available
push( eax );
call fileio.puti16;

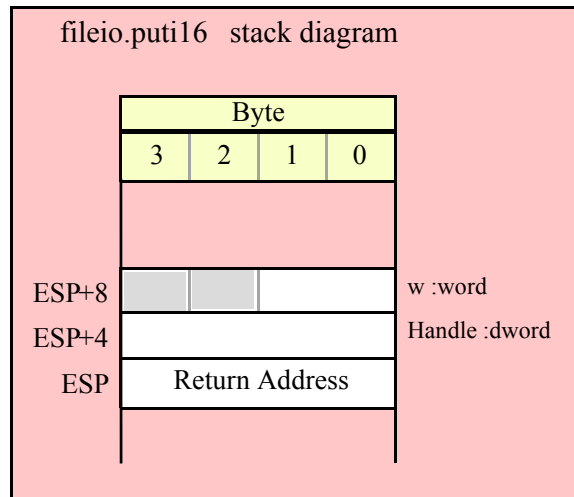
// If no register is available, do something
// like the following code:

push( fileHandle );
sub( 4, esp );
push( eax );
movzx( wordVar, eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.puti16;

// If the word value is in a 16-bit register
// then you can use code like the following:

push( fileHandle );
push( eax ); // Assume wordVar is in AX
call fileio.puti16;

```



**fileio.puti16Size( Handle:dword; w:word; width:int32; fill:char )**

This function writes the 16-bit signed integer value you pass to the specified output file using the width and fill values as specified above.

HLA high-level calling sequence examples:

```
fileio.puti16Size( fileHandle, wordVar, width, padChar );
```

HLA low-level calling sequence examples:

```
// If "wordVar" and "padChar" are not one of the last three
// bytes on a page of memory, you can do this:
```

```
push( fileHandle );
push( (type dword wordVar) );
push( width );
push( (type dword padChar) );
call fileio.puti16Size;
```

```
// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:
```

```
push( fileHandle );
movzx( wordVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call fileio.puti16Size;
```

```
// If no registers are available, do something
// like the following code:
```

```
push( fileHandle );
```

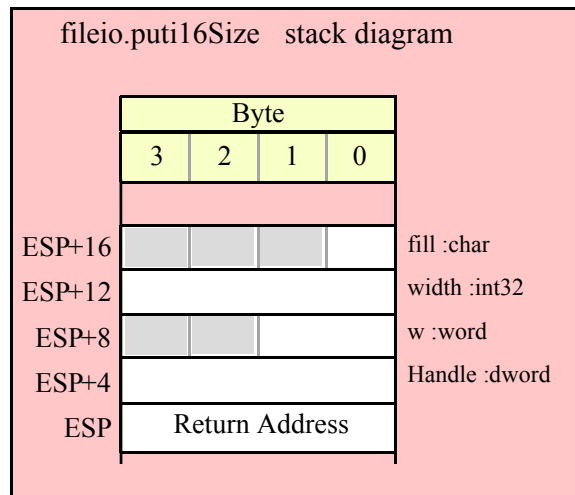
```

sub( 12, esp );
push( eax );
movzx( wordVar, eax );
mov( eax, [esp+12] );
mov( width, eax );
mov( eax, [esp+8] );
movzx( padChar, eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.puti16Size;

// If "wordVar" is in a 16-bit register
// and "padChar" is in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:

push( fileHandle );
push( eax );    // Assume wordVar is in AX
push( width );
push( ebx );    // Assume padChar is in BL
call fileio.puti16Size;

```



#### **fileio.puti32( Handle:dword; d:dword )**

This function converts the 32-bit signed integer you pass as a parameter to a string and writes this string to the file (specified by Handle) using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```

fileio.puti32( fileHandle, dwordVar );

// If the dword is in a register (EAX):

fileio.puti32( fileHandle, eax );

```

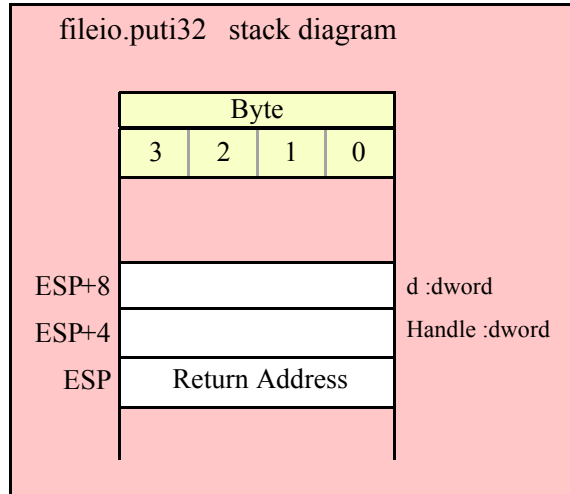
HLA low-level calling sequence examples:

```

push( fileHandle );
push( dwordVar );
call fileio.puti32;

push( fileHandle );
push( eax );
call fileio.puti32;

```



**fileio.puti32Size( Handle:dword; d:dword; width:int32; fill:char )**

This function writes the 32-bit value you pass as a signed integer to the specified output file using the width and fill values as specified above.

HLA high-level calling sequence examples:

```

fileio.puti32Size( fileHandle, dwordVar, width, ' ' );

// If the dword is in a register (EAX):

fileio.puti32Size( fileHandle, eax, width, cl );

```

HLA low-level calling sequence examples:

```

push( fileHandle );
push( dwordVar );
push( width );
pushd( ' ' );
call fileio.puti32Size;

push( fileHandle );
push( eax );
push( width );
push( ecx ); // fill char is in CL
call fileio.puti32Size;

// Assume fill char is in CH

push( fileHandle );
push( eax );

```

```

push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call fileio.puti32Size;

// Alternate method of the above

push( fileHandle );
push( eax );
push( width );
sub( 4, esp );
mov( ch, [esp] );
call fileio.puti32Size;

// If the fill char is a variable and
// a register is available, try this code:

push( fileHandle );
push( eax );
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call fileio.puti32Size;

// If the fill char is a variable and
// no register is available, here's one
// possibility:

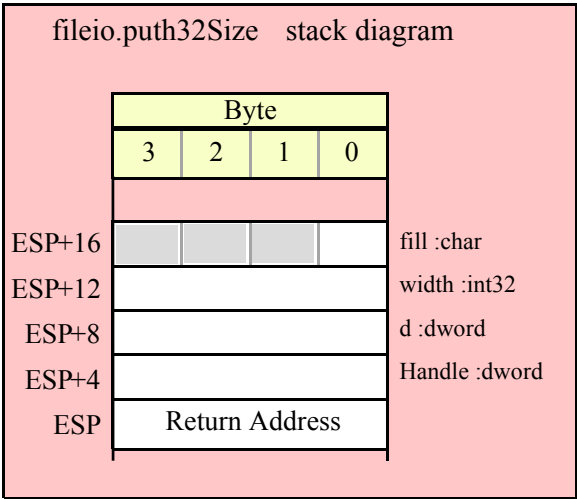
push( fileHandle );
push( eax );
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call fileio.puti32Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( fileHandle );
push( eax );
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.puti32Size;

```





**fileio.puti64( Handle:dword; q:qword )**

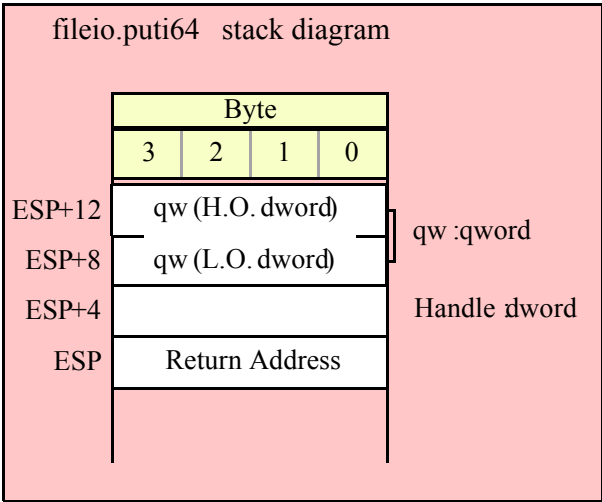
This function converts the 64-bit signed integer you pass as a parameter to a string and writes this string to the file (specified by Handle) using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
fileio.puti64( fileHandle, qwordVar );
```

HLA low-level calling sequence examples:

```
push( fileHandle );  
push( (type dword qwordVar[4]) ); // H.O. dword first  
push( (type dword qwordVar));    // L.O. dword last  
call fileio.puti64;
```



**fileio.puti64Size( Handle:dword; q:qword; width:int32; fill:char )**

This function writes the 64-bit value you pass as a signed integer to the specified output file using the width and fill values as specified above.

HLA high-level calling sequence examples:

```
fileio.puti64Size( fileHandle, qwordVar, width, ' ' );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
pushd( ' ' );
call fileio.puti64Size;
```

```
push( fileHandle );
push( edx ); // Assume 64-bit value in edx:eax
push( eax );
push( width );
push( ecx ); // fill char is in CL
call fileio.puti64Size;
```

// Assume fill char is in CH

```
push( fileHandle );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call fileio.puti64Size;
```

// Alternate method of the above

```
push( fileHandle );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
sub( 4, esp );
mov( ch, [esp] );
call fileio.puti64Size;
```

// If the fill char is a variable and  
// a register is available, try this code:

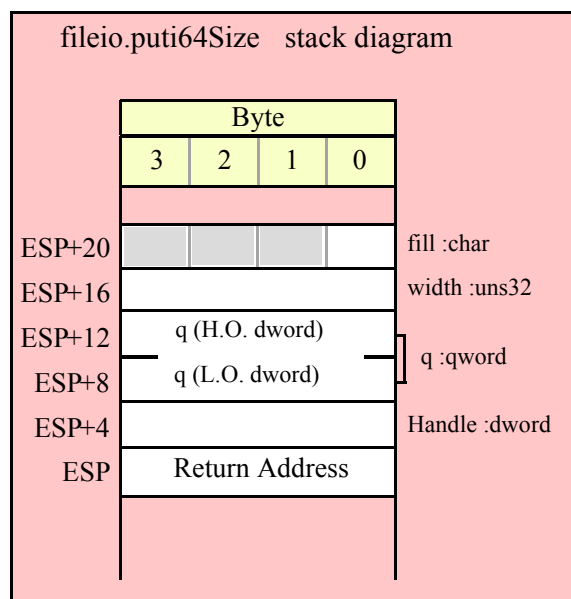
```
push( fileHandle );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call fileio.puti64Size;
```

```
// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( fileHandle );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));      // L.O. dword last
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call fileio.puti64Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( fileHandle );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));      // L.O. dword last
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.puti64Size;
```



```
fileio.puti128( Handle:dword; l:1word )
```

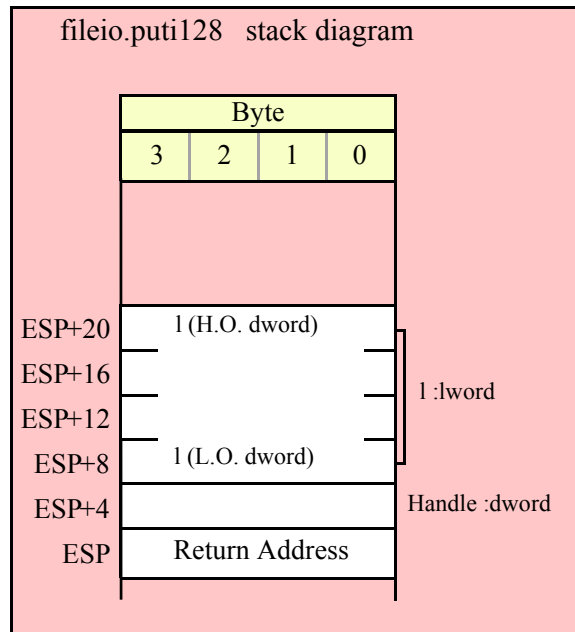
This function converts the 128-bit signed integer you pass as a parameter to a string and writes this string to the file (specified by Handle) using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
fileio.puti128( fileHandle, lwordVar );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
push( (type dword lwordVar[12]) ); // H.O. dword first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
call fileio.puti128;
```



**fileio.puti128Size( Handle:dword; l:lword; width:int32; fill:char )**

This function writes the 128-bit value you pass as a signed integer to the specified output file using the width and fill values as specified above.

HLA high-level calling sequence examples:

```
fileio.puti128Size( fileHandle, lwordVar, width, ' ' );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
push( width );
pushd( ' ' );
call fileio.puti128Size;
```

```
// Assume fill char is in CH
```

```
push( fileHandle );
push( (type dword lwordVar[12])); // Push H.O. word first
```

```

push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call fileio.puti128Size;

// Alternate method of the above

push( fileHandle );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
push( width );
sub( 4, esp );
mov( ch, [esp] );
call fileio.puti128Size;

// If the fill char is a variable and
// a register is available, try this code:

push( fileHandle );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call fileio.puti128Size;

// If the fill char is a variable and
// no register is available, here's one
// possibility:

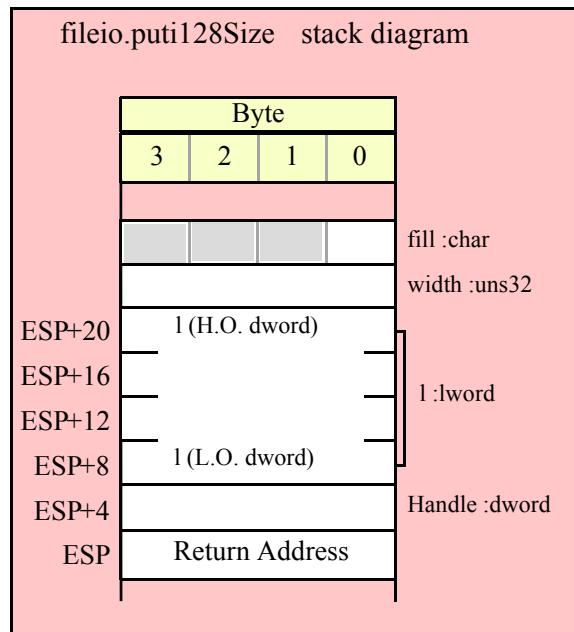
push( fileHandle );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
push( width );
push( (type dword fillChar) );      // Chance of page crossing!
call fileio.puti128Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( fileHandle );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );

```

```
pop( eax );
call fileio.puti128Size;
```



### 15.3.5 Unsigned Integer Output Routines

These routines convert unsigned integer values to string format and write that string to the file specified by the Handle parameter. The fileio.putxxxSize functions contain width and fill parameters that let you specify the minimum field width when outputting a value.

If the absolute value of width is greater than the number of print positions the value requires, then these functions output width characters to the output file. If width is non-negative, then these functions right-justify the value in the output field; if value is negative, then these functions left-justify the value in the output field.

These functions print the fill character as the padding value for the extra print positions.

```
xxxSize( value, width, fill );
```

Assuming “value” requires five print positions, “width” is eight, and fill is “f” then the xxxSize functions produce the string

f	f	f	V	A	L	U	E
---	---	---	---	---	---	---	---

Assuming “value” requires five print positions, “width” is minus eight, and fill is “f” then the xxxSize functions produce the string

V	A	L	U	E	f	f	f
---	---	---	---	---	---	---	---

```
fileio.putu8 ( Handle:dword; b:byte )
```

This function converts the eight-bit unsigned integer you pass as a parameter to a string and writes this string to the file (specified by Handle) using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
fileio.putu8( fileHandle, byteVar );
```

```
// If the character is in a register (AL):
```

```
fileio.putu8( fileHandle, al );
```

HLA low-level calling sequence examples:

```
// If "byteVar " is not one of the last three
// bytes on a page of memory, you can do this:
```

```
push( fileHandle );
push( (type dword byteVar ) );
call fileio.putu8;
```

```
// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:
```

```
push( fileHandle );
movzx( byteVar , eax ); // Assume EAX is available
push( eax );
call fileio.putu8;
```

```
// If no register is available, do something
// like the following code:
```

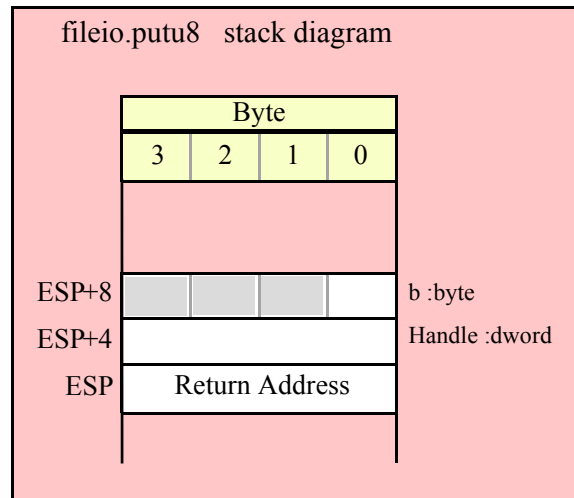
```
push( fileHandle );
sub( 4, esp );
push( eax );
movzx( byteVar , eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.putu8;
```

```
// If the character value is in al, bl, cl, or dl
// then you can use code like the following:
```

```
push( fileHandle );
push( eax ); // Assume byteVar is in AL
call fileio.putu8;
```

```
// If the byte value is in ah, bh, ch, or dh
// you'll have to use code like the following:
```

```
push( fileHandle );
xchg( al, ah ); // Assume byteVar is in AH
push( eax );    // It's now in AL
xchg( al, ah ); // Restore al/ah
call fileio.putu8;
```



**fileio.putu8Size( Handle:dword; b:byte; width:int32; fill:char )**

This function writes the unsigned eight-bit value you pass to the specified output file using the width and fill values as specified above.

HLA high-level calling sequence examples:

```
fileio.putu8Size( fileHandle, byteVar, width, padChar );
```

HLA low-level calling sequence examples:

```
// If "byteVar" and "padChar" are not one of the last three
// bytes on a page of memory, you can do this:

push( fileHandle );
push( (type dword byteVar) );
push( width );
push( (type dword padChar) );
call fileio.putu8Size;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

push( fileHandle );
movzx( byteVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call fileio.putu8Size;

// If no registers are available, do something
// like the following code:

push( fileHandle );
sub( 12, esp );
push( eax );
```



```

movzx( byteVar, eax );
mov( eax, [esp+12] );
mov( width, eax );
mov( eax, [esp+8] );
movzx( padChar, eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.putu8Size;

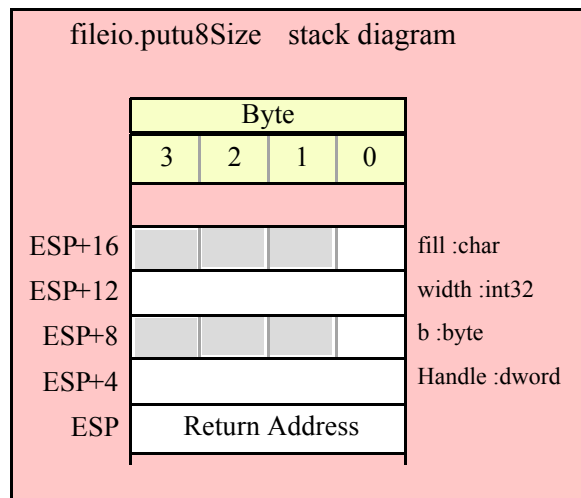
// If "byteVar" or "padChar" are in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:

push( fileHandle );
push( eax );    // Assume byteVar is in AL
push( width );
push( ebx );    // Assume padChar is in BL
call fileio.putu8Size;

// Do the following if the characters are
// in AH, BH, CH, or DH:

push( fileHandle );
xchg( al, ah );    // Assume byteVar is in AH
xchg( bl, bh );    // Assume padChar is in BH
push( eax );
push( width );
push( ebx );
xchg( al, ah );
xchg( bl, bh );
call fileio.putu8Size;

```



**fileio.putu16( Handle:dword; w:word )**

This function converts the 16-bit unsigned integer you pass as a parameter to a string and writes this string to the file (specified by Handle) using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
fileio.putu16( fileHandle, wordVar );
```

```
// If the word is in a register (AX):
```

```
fileio.putu16( fileHandle, ax );
```

HLA low-level calling sequence examples:

```
// If "wordVar " is not one of the last three  
// bytes on a page of memory, you can do this:
```

```
push( fileHandle );  
push( (type dword wordVar) );  
call fileio.putu16;
```

```
// If you can't guarantee that the previous code  
// won't generate an illegal memory access, and a  
// 32-bit register is available, use code like  
// the following:
```

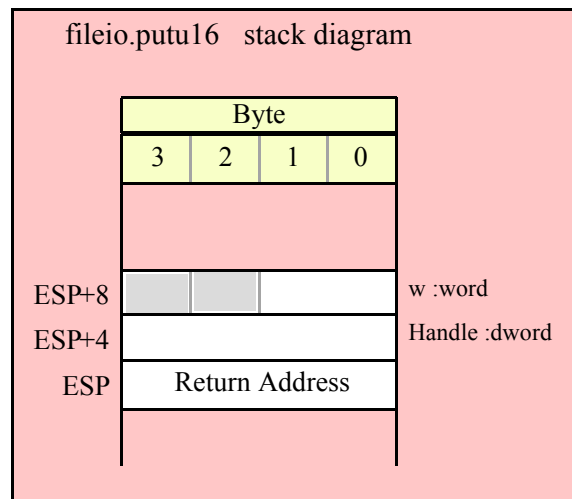
```
push( fileHandle );  
movzx( wordVar, eax ); // Assume EAX is available  
push( eax );  
call fileio.putu16;
```

```
// If no register is available, do something  
// like the following code:
```

```
push( fileHandle );  
sub( 4, esp );  
push( eax );  
movzx( wordVar, eax );  
mov( eax, [esp+4] );  
pop( eax );  
call fileio.putu16;
```

```
// If the word value is in a 16-bit register  
// then you can use code like the following:
```

```
push( fileHandle );  
push( eax ); // Assume wordVar is in AX  
call fileio.putu16;
```



**fileio.putu16Size( Handle:dword; w:word; width:int32; fill:char )**

This function writes the unsigned 16-bit value you pass to the specified output file using the width and fill values as specified above.

HLA high-level calling sequence examples:

```
fileio.putu16Size( fileHandle, wordVar, width, padChar );
```

HLA low-level calling sequence examples:

```
// If "wordVar" and "padChar" are not one of the last three
// bytes on a page of memory, you can do this:
```

```
push( fileHandle );
push( (type dword wordVar) );
push( width );
push( (type dword padChar) );
call fileio.putu16Size;
```

```
// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:
```

```
push( fileHandle );
movzx( wordVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call fileio.putu16Size;
```

```
// If no registers are available, do something
// like the following code:
```

```
push( fileHandle );
sub( 12, esp );
```

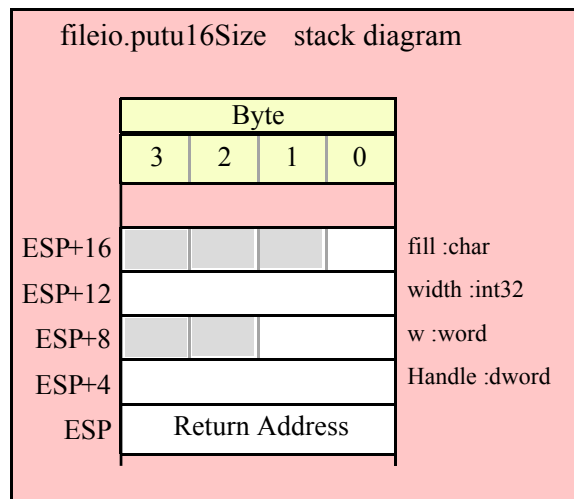
```

push( eax );
movzx( wordVar, eax );
mov( eax, [esp+12] );
mov( width, eax );
mov( eax, [esp+8] );
movzx( padChar, eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.putul6Size;

// If "wordVar" is in a 16-bit register
// and "padChar" is in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:

push( fileHandle );
push( eax );    // Assume wordVar is in AX
push( width );
push( ebx );    // Assume padChar is in BL
call fileio.putul6Size;

```



### fileio.putu32( Handle:dword; d:dword )

This function converts the 32-bit unsigned integer you pass as a parameter to a string and writes this string to the file (specified by Handle) using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```

fileio.putu32( fileHandle, dwordVar );

// If the dword is in a register (EAX):

fileio.putu32( fileHandle, eax );

```

HLA low-level calling sequence examples:

```

push( fileHandle );

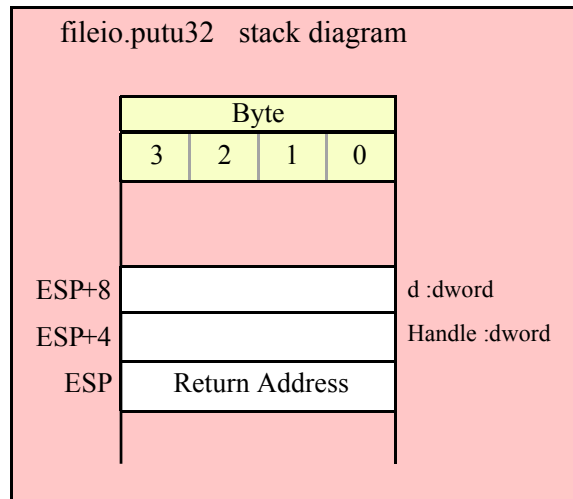
```

```

push( dwordVar );
call fileio.putu32;

push( fileHandle );
push( eax );
call fileio.putu32;

```



**fileio.putu32Size( Handle:dword; d:dword; width:int32; fill:char )**

This function writes the unsigned 32-bit value you pass to the specified output file using the width and fill values as specified above.

HLA high-level calling sequence examples:

```

fileio.putu32Size( fileHandle, dwordVar, width, ' ' );

// If the dword is in a register (EAX):

fileio.putu32Size( fileHandle, eax, width, cl );

```

HLA low-level calling sequence examples:

```

push( fileHandle );
push( dwordVar );
push( width );
pushd( ' ' );
call fileio.putu32Size;

push( fileHandle );
push( eax );
push( width );
push( ecx ); // fill char is in CL
call fileio.putu32Size;

// Assume fill char is in CH

push( fileHandle );
push( eax );

```

```

push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call fileio.putu32Size;

// Alternate method of the above

push( fileHandle );
push( eax );
push( width );
sub( 4, esp );
mov( ch, [esp] );
call fileio.putu32Size;

// If the fill char is a variable and
// a register is available, try this code:

push( fileHandle );
push( eax );
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call fileio.putu32Size;

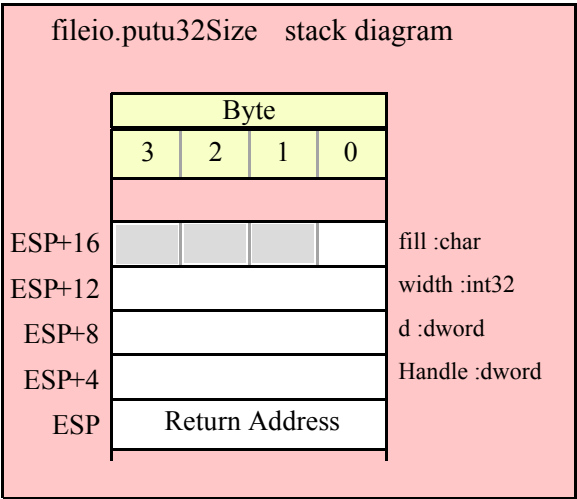
// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( fileHandle );
push( eax );
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call fileio.putu32Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( fileHandle );
push( eax );
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.putu32Size;

```



**fileio.putu64( Handle:dword; q:qword )**

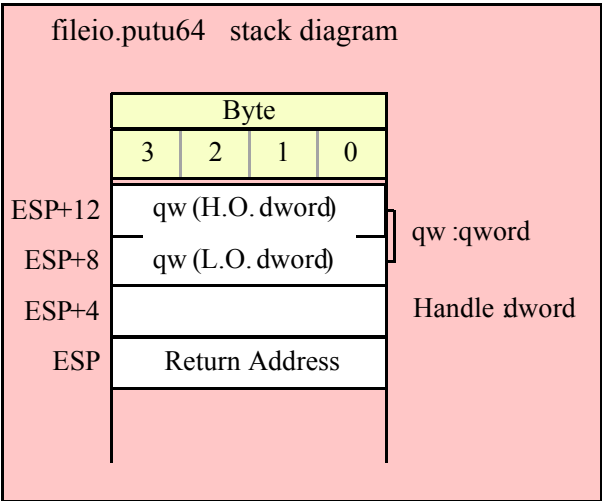
This function converts the 64-bit unsigned integer you pass as a parameter to a string and writes this string to the file (specified by Handle) using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
fileio.putu64( fileHandle, qwordVar );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
call fileio.putu64;
```



**fileio.putu64Size( Handle:dword; q:qword; width:int32; fill:char )**

This function writes the unsigned 64-bit value you pass to the specified output file using the width and fill values as specified above.

HLA high-level calling sequence examples:

```
fileio.putu64Size( fileHandle, qwordVar, width, ' ' );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
pushd( ' ' );
call fileio.putu64Size;
```

```
push( fileHandle );
push( edx ); // Assume 64-bit value in edx:eax
push( eax );
push( width );
push( ecx ); // fill char is in CL
call fileio.putu64Size;
```

// Assume fill char is in CH

```
push( fileHandle );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call fileio.putu64Size;
```

// Alternate method of the above

```
push( fileHandle );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
sub( 4, esp );
mov( ch, [esp] );
call fileio.putu64Size;
```

// If the fill char is a variable and  
// a register is available, try this code:

```
push( fileHandle );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call fileio.putu64Size;
```



```

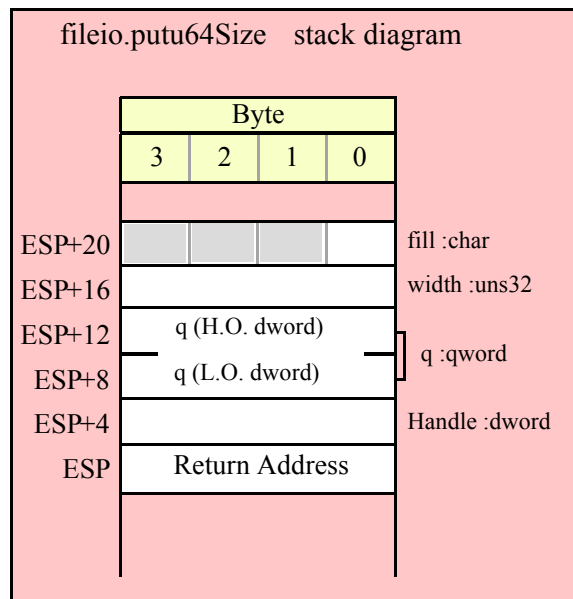
// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( fileHandle );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call fileio.putu64Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( fileHandle );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call fileio.putu64Size;

```



#### **fileio.putu128( Handle:dword; l:lword )**

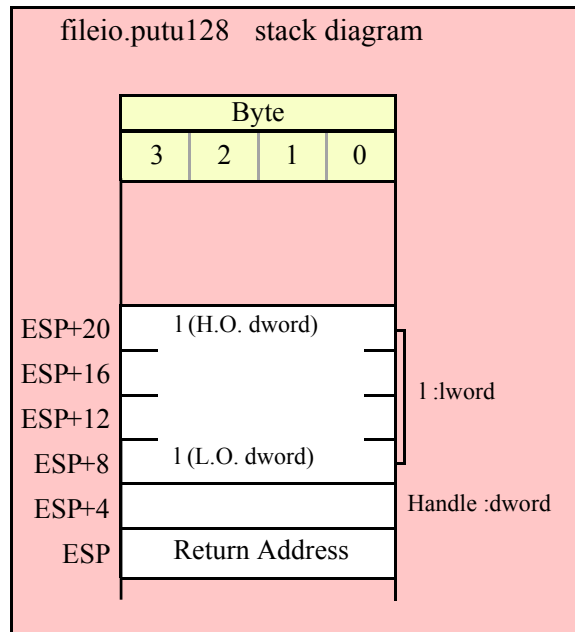
This function converts the 128-bit unsigned integer you pass as a parameter to a string and writes this string to the file (specified by Handle) using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
fileio.putu128( fileHandle, lwordVar );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
push( (type dword lwordVar[12]) ); // H.O. dword first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
call fileio.putu128;
```



**fileio.putu128Size( Handle:dword; l:lword; width:int32; fill:char )**

This function writes the unsigned 128-bit value you pass to the specified output file using the width and fill values as specified above.

HLA high-level calling sequence examples:

```
fileio.putu128Size( fileHandle, lwordVar, width, ' ' );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));    // L.O. dword last
push( width );
pushd( ' ' );
call fileio.putu128Size;
```

```
// Assume fill char is in CH
```

```
push( fileHandle );
```

```

push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar)); // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call fileio.putul28Size;

// Alternate method of the above

push( fileHandle );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar)); // L.O. dword last
push( width );
sub( 4, esp );
mov( ch, [esp] );
call fileio.putul28Size;

// If the fill char is a variable and
// a register is available, try this code:

push( fileHandle );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar)); // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call fileio.putul28Size;

// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( fileHandle );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar)); // L.O. dword last
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call fileio.putul28Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

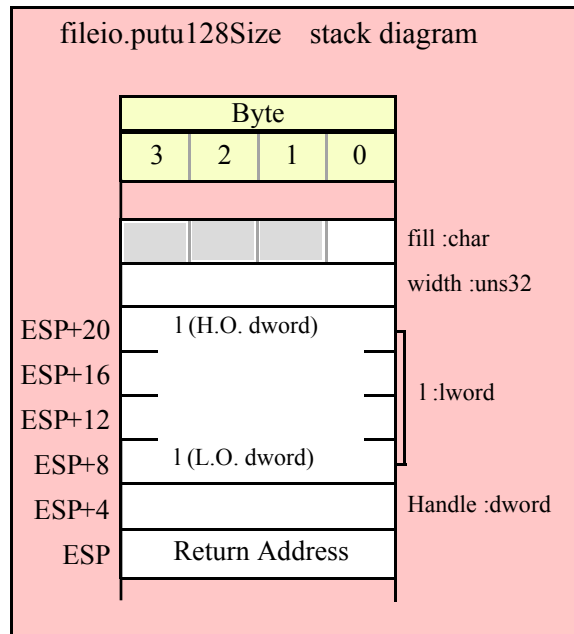
push( fileHandle );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar)); // L.O. dword last
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );

```

```

mov( eax, [esp+4] );
pop( eax );
call fileio.putu128Size;

```



### 15.3.6 Floating Point Output Routines

The HLA file I/O class provides several procedures you can use to write floating point files to a text file. The following subsections describe these routines.

#### 15.3.6.1 Real Output Using Scientific Notation

The floating point numeric output routines translate the three different binary floating point formats to their string representation and then write this string to the file that the Handle parameter specifies. There are two generic classes of these routines: those that convert their values to exponential/scientific notation and those that convert their string to a decimal form.

The fileio.pute80, fileio.pute64, and fileio.pute32 routines convert their values to a string using scientific notation. These three routines each have two parameters: the value to output and the field width of the result. These routines produce a string with the following format:

s	i	.	f	f	f	f	f	E	±	x
---	---	---	---	---	---	---	---	---	---	---

s is a space for positive values, '-' for negative values  
i represents the integer portion of the mantissa  
ffff represents the fractional portion of the mantissa  
x is one or more base-10 exponent digits.

```
fileio.pute32( Handle:dword; r:real32; width:uns32 )
```

This function writes the 32-bit single precision floating point value passed in r to the file using scientific/exponential notation. This procedure prints the value using width print positions in the file. width should have a

minimum value of five for real numbers in the range 1e-9..1e+9 and a minimum value of six for all other values. Note that 32-bit extended precision floating point values support about 6-7 significant digits. So a width value that yields more than seven mantissa digits will produce garbage output in the low order digits of the number.

HLA high-level calling sequence examples:

```
fileio.put32( fileHandle, r32Var, width );

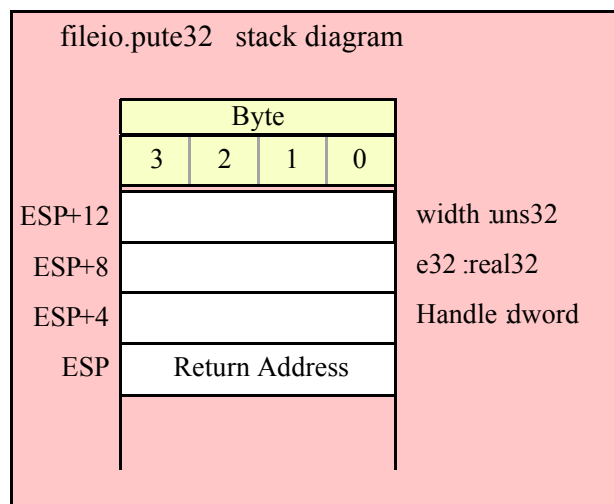
// If the real32 value is in an FPU register (ST0):

var
  r32Temp:real32;
  .
  .
  .
fstp( r32Temp );
fileio.put32( fileHandle, r32Temp, 12 );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
push( (type dword r32Var) );
push( width );
call fileio.put32;

push( fileHandle );
sub( 4, esp );
fstp( (type real32 [esp]) );
pushd( 12 );
call fileio.put32;
```



**fileio.put64( Handle:dword; r:real64; width:uns32 )**

This function writes the 64-bit double precision floating point value passed in r to the file using scientific/exponential notation. This procedure prints the value using width print positions in the file. width should have a minimum value of five for real numbers in the range 1e-9..1e+9 and a minimum value of six for all other values. Note that 64-bit double precision floating point values support about 15 significant digits. So a width value that yields more than 15 mantissa digits will produce garbage output in the low order digits of the number.

HLA high-level calling sequence examples:

```
fileio.pute64( fileHandle, r64Var, width );

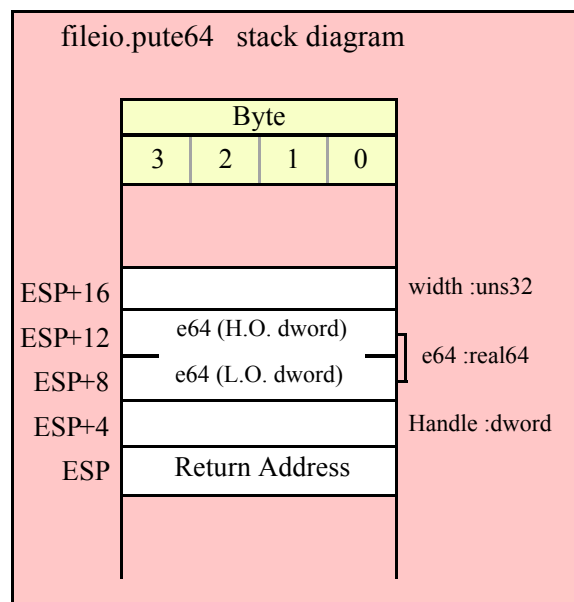
// If the real64 value is in an FPU register (ST0):

var
    r64Temp:real64;
    .
    .
    .
fstp( r64Temp );
fileio.pute64( fileHandle, r64Temp, 12 );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
push( (type dword r64Var[4]));
push( (type dword r64Var[0]));
push( width );
call fileio.pute64;

push( fileHandle );
sub( 8, esp );
fstp( (type real64 [esp]) );
pushd( 12 );
call fileio.pute64;
```



**fileio.pute80( Handle:dword; r:real80; width:uns32 )**

This function writes the 80-bit extended precision floating point value passed in *r* to the file using scientific/exponential notation. This procedure prints the value using *width* print positions in the file. *width* should have a minimum value of five for real numbers in the range 1e-9..1e+9 and a minimum value of six for all other values. Note that 80-bit extended precision floating point values support about 18 significant digits. So a *width* value that yields more than 18 mantissa digits will produce garbage output in the low order digits of the number.

HLA high-level calling sequence examples:

```
fileio.pute80( fileHandle, r80Var, width );

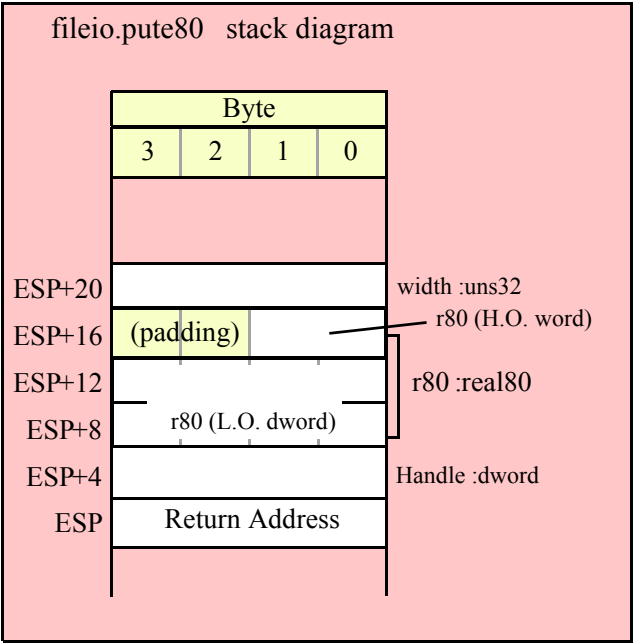
// If the real80 value is in an FPU register (ST0):

var
    r80Temp:real80;
    .
    .
    .
fstp( r80Temp );
fileio.pute80( fileHandle, r80Temp, 12 );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
pushw( 0 ); // A word of padding.
push( (type word r80Var[8]));
push( (type dword r80Var[4]));
push( (type dword r80Var[0]));
push( width );
call fileio.pute80;

push( fileHandle );
sub( 12, esp );
fstp( (type real80 [esp]) );
pushd( 12 );
call fileio.pute80;
```



### 15.3.6.2 Real Output Using Decimal Notation

Although scientific (exponential) notation is the most general display format for real numbers, real numbers you display in this format are very difficult to read. Therefore, the HLA fileio module also provides a set of functions that output real values using the decimal representation. Although you cannot (practically) use these decimal output routines for all real values, they are applicable to a wide variety of common numbers you will use in your programs.

These functions come in two varieties. The first variety requires five parameters: the real value to convert, the width of the converted value, the number of digit positions to the right of the decimal point, and a padding character. The second variety only requires the first four parameters and assumes the padding character is a space. These functions write their values using the following string format:

s	i	i	i	.	f	f	f	f	f	f
---	---	---	---	---	---	---	---	---	---	---

s is a space for positive values, '-' for negative values  
i represents the integer portion of the mantissa  
ffffff represents the fractional portion of the mantissa

**fileio.putr32( Handle:dword; r:real32; width:uns32; decpts:uns32; pad:char )**

This procedure writes a 32-bit single precision floating point value to the filevar file as a string. The string consumes exactly width characters in the output file. If the numeric output, using the specified number of positions to the right of the decimal point, is sufficiently small that the string representation would be less than width characters, then this procedure uses the value of pad as the padding character to fill the output with width characters. The number is right-justified within the output field (that is, this function prints the padding characters before the string representation of the number).

HLA high-level calling sequence examples:

```
fileio.putr32( fileHandle, r32Var, width, decpts, fill );
fileio.putr32( fileHandle, r32Var, 10, 2, '*' );
```

// If the real32 value is in an FPU register (ST0):

```
var
    r32Temp:real32;
.
.
.
fstp( r32Temp );
fileio.putr32( fileHandle, r32Temp, 12, 2, al );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
push( (type dword r32Var) );
push( width );
push( decpts );
movzx( fill, eax );
push( eax );
call fileio.putr32;
```

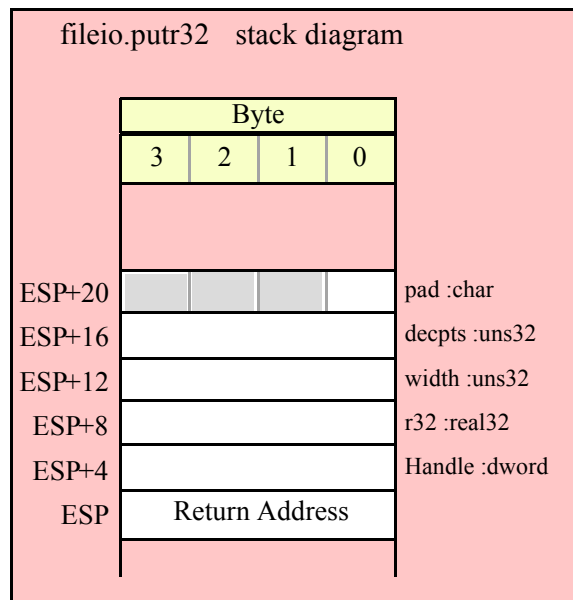


```

push( fileHandle );
push( (type dword r32Var) );
push( width );
push( decpts );
pushd( (type dword fill) ); // If no memory fault possible
call fileio.putr32;

push( fileHandle );
sub( 4, esp );
fstp( (type real32 [esp]) );
pushd( 12 );
sub( 4, esp );
push( eax );
movzx( fill, eax );// If memory fault were possible
mov( eax, [esp+4] ); // in above code.
pop( eax );
call fileio.putr32;

```



**fileio.putr64( Handle:dword; r:real64; width:uns32; decpts:uns32; pad:char )**

This procedure writes a 64-bit double precision floating point value to the file as a string. The string consumes exactly width characters in the output file. If the numeric output, using the specified number of positions to the right of the decimal point, is sufficiently small that the string representation would be less than width characters, then this procedure uses the value of pad as the padding character to fill the output with width characters.

HLA high-level calling sequence examples:

```

fileio.putr64( fileHandle, r64Var, width, decpts, fill );
fileio.putr64( fileHandle, r64Var, 10, 2, '*' );

```

// If the real64 value is in an FPU register (ST0):

```

var
  r64Temp:real64;
  .
  .

```

```

    .
    fstp( r64Temp );
    fileio.putr64( fileHandle, r64Temp, 12, 2, al );

```

HLA low-level calling sequence examples:

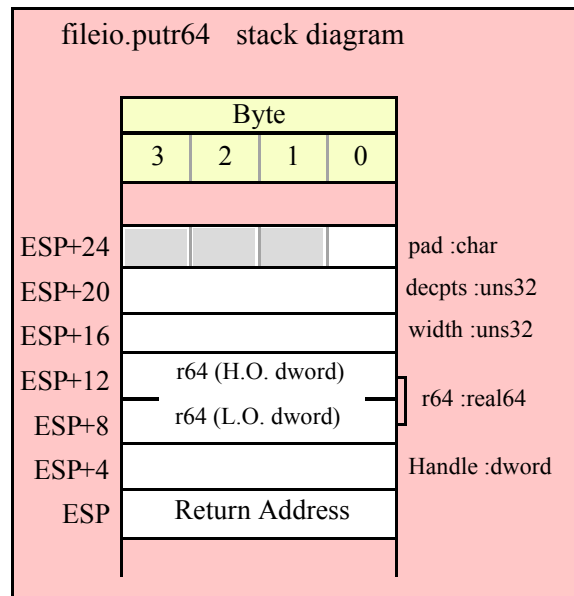
```

push( fileHandle );
push( (type dword r64Var[4]) );
push( (type dword r64Var) );
push( width );
push( decpts );
movzx( fill, eax );
push( eax );
call fileio.putr64;

push( fileHandle );
push( (type dword r64Var[4]) );
push( (type dword r64Var) );
push( width );
push( decpts );
pushd( (type dword fill) ); // If no memory fault possible
call fileio.putr64;

push( fileHandle );
sub( 8, esp );
fstp( (type real64 [esp]) );
pushd( 12 );
sub( 4, esp );
push( eax );
movzx( fill, eax ); // If memory fault were possible
mov( eax, [esp+4] ); // in above code.
pop( eax );
call fileio.putr64;

```



**fileio.putr80( Handle:dword; r:real80; width:uns32; decpts:uns32; pad:char )**

This procedure writes an 80-bit extended precision floating point value to the file as a string. The string consumes exactly width characters in the output file. If the numeric output, using the specified number of positions to the right of the decimal point, is sufficiently small that the string representation would be less than width characters, then this procedure uses the value of pad as the padding character to fill the output with width characters.

HLA high-level calling sequence examples:

```
fileio.putr80( fileHandle, r80Var, width, decpts, fill );
fileio.putr80( fileHandle, r80Var, 10, 2, '*' );
```

// If the real80 value is in an FPU register (ST0):

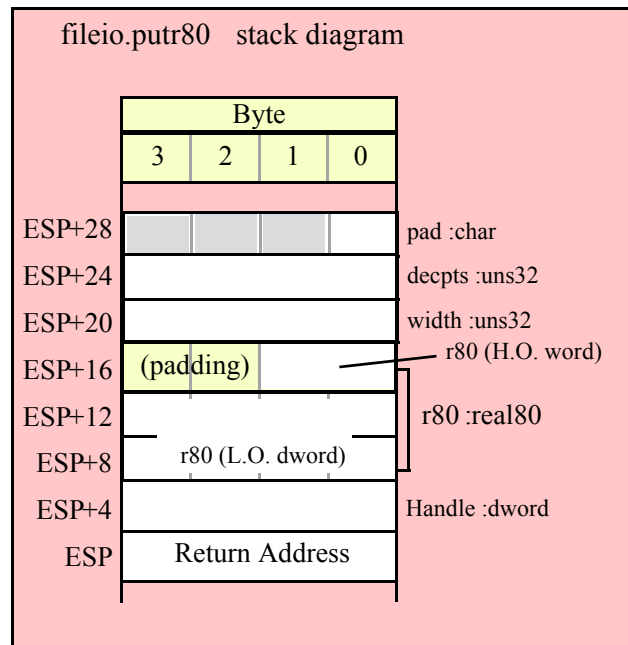
```
var
    r80Temp:real80;
    .
    .
    .
fstp( r80Temp );
fileio.putr80( fileHandle, r80Temp, 12, 2, al );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
pushw( 0 );
push( (type word r80Var[8]) );
push( (type dword r80Var[4]) );
push( (type dword r80Var) );
push( width );
push( decpts );
movzx( fill, eax );
push( eax );
call fileio.putr80;

push( fileHandle );
pushw( 0 );
push( (type word r80Var[8]) );
push( (type dword r80Var[4]) );
push( (type dword r80Var) );
push( width );
push( decpts );
pushd( (type dword fill) ); // If no memory fault possible
call fileio.putr80;

push( fileHandle );
sub( 12, esp );
fstp( (type real80 [esp]) );
pushd( 12 );
sub( 4, esp );
push( eax );
movzx( fill, eax ); // If memory fault were possible
mov( eax, [esp+4] ); // in above code.
pop( eax );
call fileio.putr80;
```



### 15.3.7 Generic File Output Routine

```
fileio.put( list of items )
```

fileio.put is a macro that automatically invokes an appropriate fileio output routine based on the type of the parameter(s) you pass it. This is a very convenient output routine and is probably the fileio output call you will use most often in your programs. Keep in mind that this macro is not a single function call; instead, HLA translates this macro into a sequence of calls to procedures like fileio.putu32, fileio.puts, etc.

fileio.put is a macro that provides a flexible syntax for outputting data to the standard output device. This macro allows a variable number of parameters. For each parameter present in the list, fileio.put will call the appropriate routine to emit that data, according to the type of the parameter. Parameters may be constants, registers, or memory locations. You must separate each macro parameter with a comma.

Here is an example of a typical invocation of `fileio.put`:

```
fileio.put( fileHandle, "I=", i, " j=", j, nl );
```

The above is roughly equivalent to

```
fileio.puts( fileHandle, "I= " );
fileio.putu32( fileHandle, i );
fileio.puts( fileHandle, " j=" );
fileio.putu32( fileHandle, j );
fileio.newln(fileHandle);
```

This assumes, of course, that `i` and `j` are `int32` variables.

The `fileio.put` macro also lets you specify the minimum field width for each parameter you specify. To print a value using a minimum field width, follow the object you wish to print with a colon and the value of the minimum field width. The previous example, using field widths, could look like the following:

```
fileio.put( fileHandle, "I=", i:2, " j=", j:5, nl );
```

Although this example used the literal decimal constants two and five for the field widths, keep in mind that register values and memory value (integers, anyway) are perfectly legal here.

For floating point numbers you wish to display in decimal form, you can specify both the minimum field width and the number of digits to print to the right of the decimal point by using the following syntax:

```
fileio.put( fileHandle, "Real value is ", f:10:3, nl );
```

The `fileio.put` macro can handle all the basic primitive types, including boolean, unsigned (8, 16, 32, 64, 128), signed (8, 16, 32, 64, 128), character, character set, real (32, 64, 80), string, and hexadecimal (byte, word, dword, qword, lword).

If you specify a class variable (object) and that class defines a "toString" method, then `fileio.put` macro will call the associated `toString` method and output that string to the file. Note that the `toString` method must dynamically allocate storage for the string by calling `stralloc`. This is because `fileio.put` will call `strfree` on the string once it outputs the string.

There is a known "design flaw" in the `fileio.put` macro. You cannot use it to print HLA intermediate variables (i.e., non-local VAR objects). The problem is that HLA's syntax for non-local accesses takes the form "reg32:varname" and `fileio.put` cannot determine if you want to print reg32 using varname print positions versus simply printing the non-local varname object. If you want to display non-local variables you must copy the non-local object into a register, a static variable, or a local variable prior to using `fileio.put` to print it. Of course, there is no problem using the other `fileio.putXXXX` functions to display non-local VAR objects, so you can use those as well.

## 15.4 File Input Routines

The HLA Standard Library provides a complementary set of file input routines. These routines behave in a fashion quite similar to the `stdin.XXXX` routines. See those routines for additional examples of these procedures.

### 15.4.1 General File Input Routines

```
fileio.read( Handle:dword; var buffer:byte; count:uns32 )
```

This routine reads a sequence of count bytes from the specified file, storing the bytes into memory at the address specified by buffer.

HLA high-level calling sequence examples:

```
fileio.read( fileHandle, buffer, count );
fileio.read( fileHandle, [eax], 1024 );
```

HLA low-level calling sequence examples:

```
// If buffer is a static variable:
```

```
push( fileHandle );
pushd( &buffer );
push( count );
call fileio.read;
```

```
// If buffer is not static, 32-bit register available:
```

```
push( fileHandle );
lea( eax, buffer );
push( eax );
push( count );
call fileio.read;
```

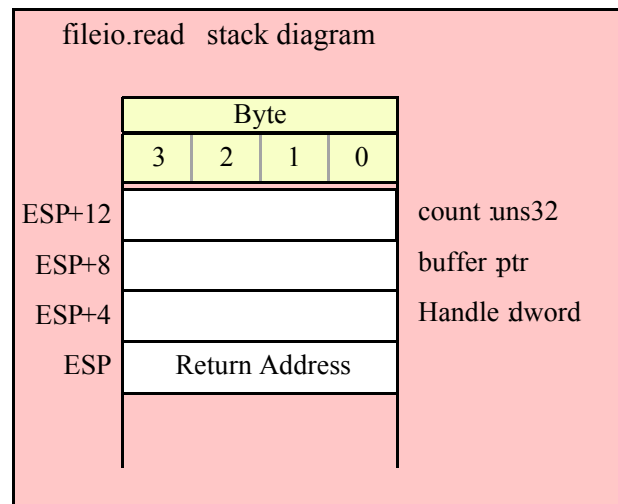
```
// If buffer is not static, no register available:
```

```
push( fileHandle );
sub( 4, esp );
```

```

push( eax );
lea( eax, buffer );
mov( eax, [esp+4] );
pop( eax );
push( count );
call fileio.read;

```



**fileio.readLn( Handle:dword );**

This function reads, and discards, all characters in the file up to the next newline sequence (or end of file).

HLA high-level calling sequence examples:

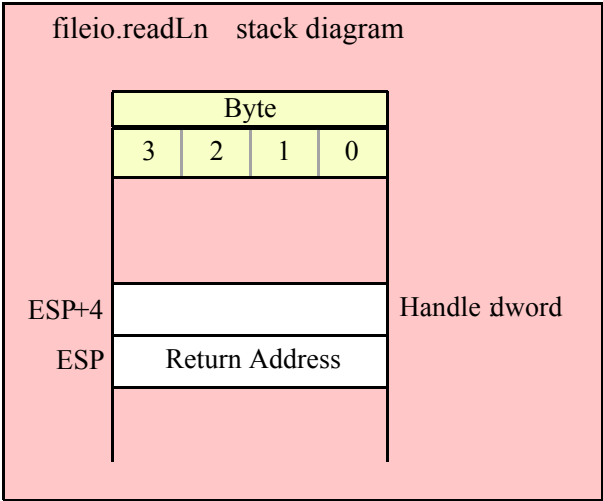
```
fileio.readLn( fileHandle );
```

HLA low-level calling sequence examples:

```

push( fileHandle );
call fileio.readLn;

```



```
fileio.eoln( Handle:dword ); @returns( "al" );
```

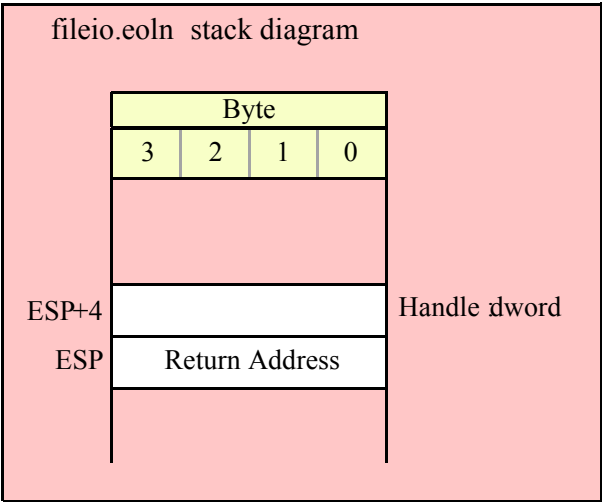
This function returns true (1) in EAX if the file is at the end of a line (note that the "returns" value is "al" even though this function returns its result in all of EAX). This function eats the newline sequence from the input.

HLA high-level calling sequence examples:

```
fileio.eoln( fileHandle );  
mov( al, eolnVar );
```

HLA low-level calling sequence examples:

```
push( fileHandle );  
call fileio.eoln;  
mov( al, eolnVar );
```

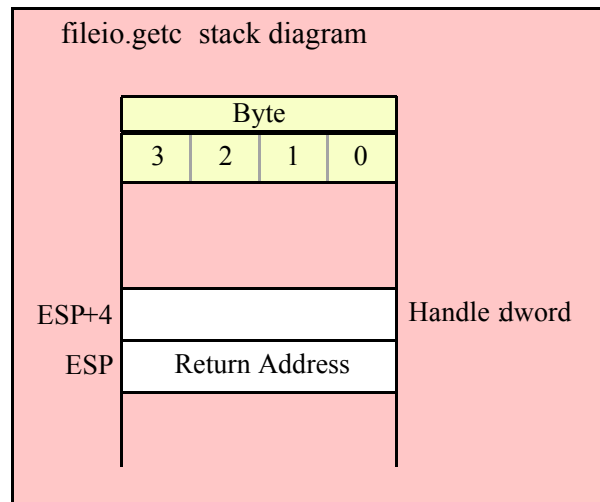


## 15.4.2 Character and String Input Routines

The following functions read character data from an input file specified by filevar. Note that HLA's fileio module does not provide the ability to read character set data directly from the user. However, you can always read a string and then convert that string to a character set using the appropriate function in the cset module.

```
fileio.getc( Handle:dword ); @returns( "al" );
```

This function reads a single character from the file and returns that character in the AL register. This function assumes that the file you've opened is a text file. Note that fileio.getc does not return the end of line sequence as part of the input stream. Use the fileio.eoln function to determine when you've reached the end of a line of text. Because fileio.getc preprocesses the text file (removing end of line sequences) you should not use it to read binary data, use it only to read text files.



```
fileio.gets( Handle:dword; s:string );
```

This function reads a sequence of characters from the current file position through to the next end of line sequence and stores these characters (without the end of line sequence) into the string variable you pass as a parameter. Before calling this routine, you must allocate sufficient storage for the string. If fileio.gets attempts to read a larger string than the string's MaxStrLen value, fileio.gets raises a string overflow exception.

Note that this function does not store the end of line sequence into the string, though it does consume the end of line sequence. The next character a fileio function will read from the file will be the first character of the following line.

If the current file position is at the end of some line of text, then fileio.gets consumes the end of line and stores the empty string into the s parameter.

HLA high-level calling sequence examples:

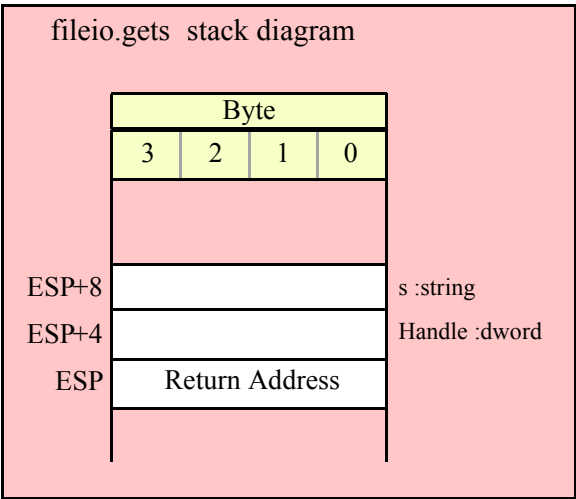
```
fileio.gets( fileHandle, inputStr );
fileio.gets( fileHandle, eax ); // EAX contains string value
```

HLA low-level calling sequence examples:

```
push( fileHandle );
push( inputStr );
call fileio.gets;
```

```
push( fileHandle );
push( eax );
call fileio.gets;
```





```
fileio.a_gets( Handle:dword ); @returns( "eax" );
```

Like fileio.gets, this function also reads a string from the file. However, rather than storing the string data into a string you supply, this function allocates storage for the string on the heap and returns a pointer to this string in the EAX register. You code should call strfree to release this storage when you're done with the string data.

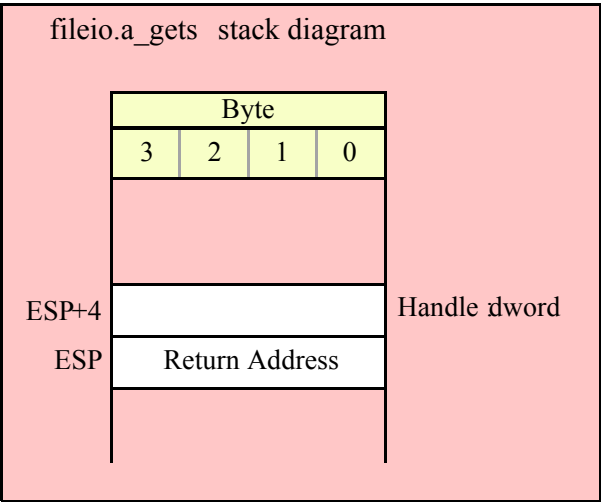
The fileio.a\_gets function imposes a line length limit of 1,024 characters. If this is a problem, you should modify the source code for this function to raise the limit. This functions raises an exception if you attempt to read a line longer than this internal limit.

HLA high-level calling sequence examples:

```
fileio.a_gets( fileHandle );  
mov( eax, inputStr );
```

HLA low-level calling sequence examples:

```
push( fileHandle );  
call fileio.a_gets;  
mov( eax, inputStr );
```



### 15.4.3 Signed Integer Input Routines

```
fileio.geti8( Handle:dword ); @returns( "al" );
```

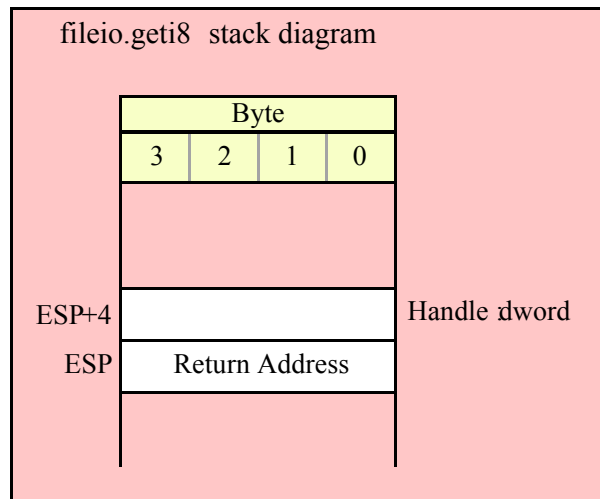
This function reads a signed eight-bit decimal integer in the range -128..+127 from the file. The number may begin with any number of delimiter characters (see the `conv.setDelimiter` and `conv.getDelimiter` functions for details on the delimiter characters) followed by an optional minus sign and a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The `fileio.geti8` function raises an appropriate exception if the input violates any of these rules or the value is outside the range -128..+127. This function returns the binary form of the integer in the AL register.

HLA high-level calling sequence examples:

```
fileio.geti8( fileHandle );
mov( al, i8Var );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
call fileio.geti8;
mov( al, i8Var );
```



```
fileio.geti16( Handle:dword ); @returns( "ax" );
```

This function reads a signed 16-bit decimal integer in the range -32768..+32767 from the file. The number may begin with any number of delimiter characters (see the `conv.setDelimiter` and `conv.getDelimiter` functions for details on the delimiter characters) followed by an optional minus sign and a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The `fileio.geti16` function raises an appropriate exception if the input violates any of these rules or the value is outside the range -32768..+32767. This function returns the binary form of the integer in the AX register.

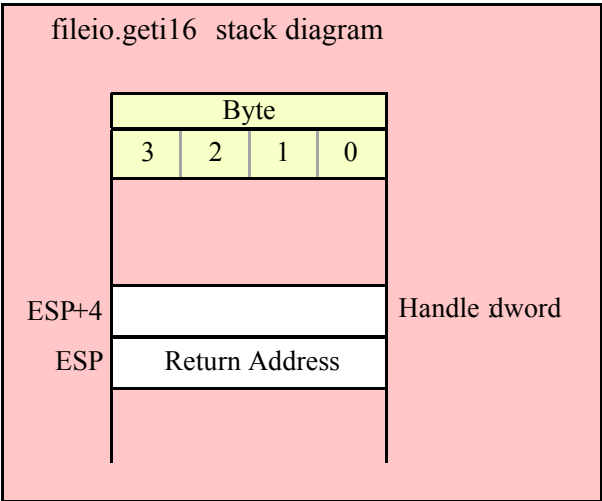
HLA high-level calling sequence examples:

```
fileio.geti16( fileHandle );
mov( ax, i16Var );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
call fileio.geti16;
```

```
mov( ax, i16Var );
```



```
fileio.geti32( Handle:dword ); @returns( "eax" );
```

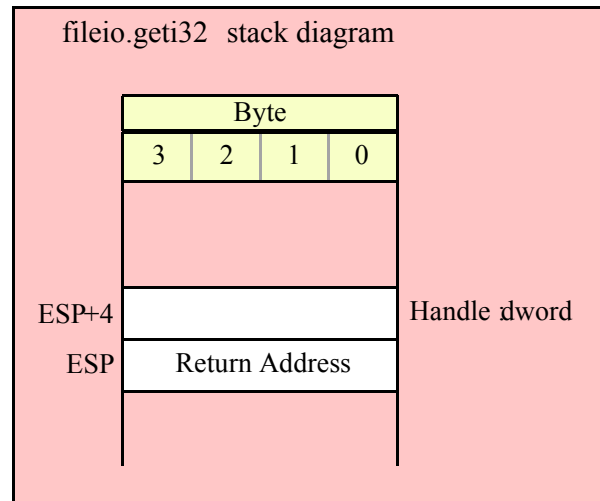
This function reads a signed 32-bit decimal integer in the (approximate) range  $\pm 2$  Billion from the file. The number may begin with any number of delimiter characters (see the `conv.setDelimiter` and `conv.getDelimiter` functions for details on the delimiter characters) followed by an optional minus sign and a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The `fileio.geti32` function raises an appropriate exception if the input violates any of these rules or the value is outside the range plus or minus two billion. This function returns the binary form of the integer in the EAX register.

HLA high-level calling sequence examples:

```
fileio.geti32( fileHandle );  
mov( eax, i32Var );
```

HLA low-level calling sequence examples:

```
push( fileHandle );  
call fileio.geti32;  
mov( eax, i32Var );
```



```
fileio.geti64( Handle:dword );
```

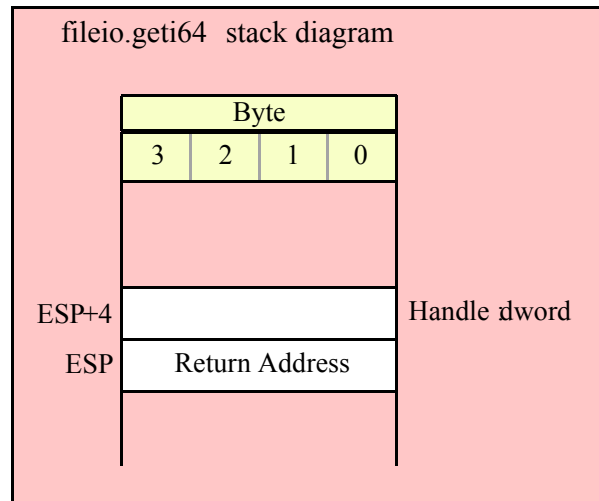
This function reads a signed 64-bit decimal integer from the file. The number may begin with any number of delimiter characters (see the `conv.setDelimiter` and `conv.getDelimiter` functions for details on the delimiter characters) followed by an optional minus sign and a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The `fileio.geti64` function raises an appropriate exception if the input violates any of these rules or the value is outside the range of a 64-bit signed integer. This function returns the 64-bit result in EDX:EAX.

HLA high-level calling sequence examples:

```
fileio.geti64( fileHandle );
mov( edx, (type dword i64Var[4]) );
mov( eax, (type dword i64Var[0]) );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
call fileio.geti64;
mov( edx, (type dword i64Var[4]) );
mov( eax, (type dword i64Var[0]) );
```



```
fileio.geti128( Handle:dword; var dest:lword );
```

This function reads a signed 128-bit decimal integer from the file. The number may begin with any number of delimiter characters (see the `conv.setDelimiter` and `conv.getDelimiter` functions for details on the delimiter characters) followed by an optional minus sign and a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The `fileio.geti128` function raises an appropriate exception if the input violates any of these rules or the value is outside the range of a 128-bit signed integer. This function stores the 128-bit result in the `lword` you pass as a reference parameter.

HLA high-level calling sequence examples:

```
fileio.geti128( fileHandle, lwordVar );
```

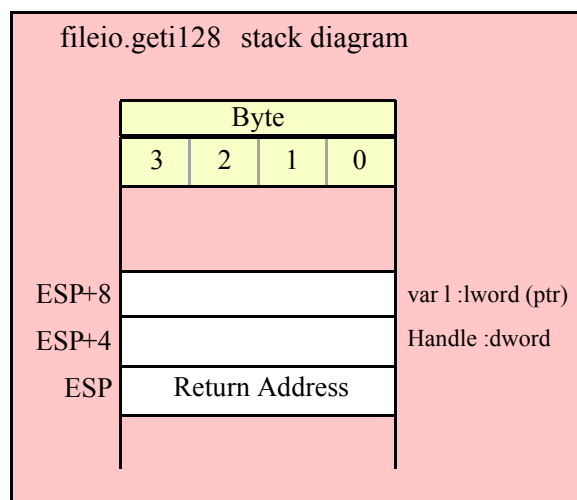
HLA low-level calling sequence examples:

```
// If lwordVar is a static variable:
```

```
push( fileHandle );
pushd( &lwordVar );
call fileio.geti128;
```

```
// If lwordVar is a not static variable
// and a 32-bit register is available:
```

```
push( fileHandle );
lea( eax, lwordVar ); // Assume EAX is available
push( eax );
call fileio.geti128;
```



## 15.4.4 Unsigned Integer Input Routines

```
fileio.getu8( Handle:dword ); @returns( "al" );
```

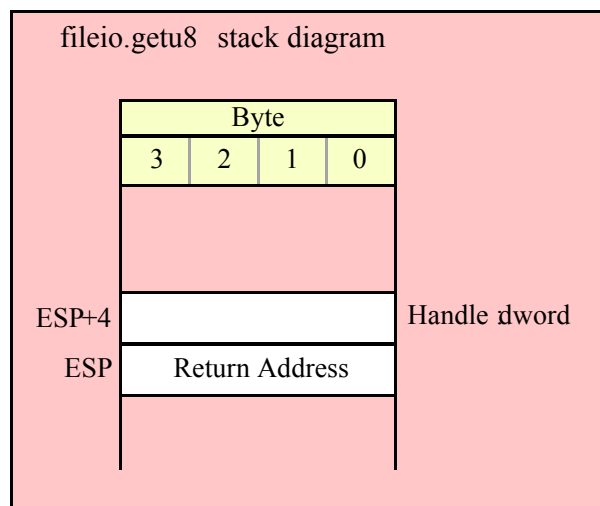
This function reads an unsigned eight-bit decimal integer in the range 0..+255 from the file. The number may begin with any number of delimiter characters (see the `conv.setDelimiter` and `conv.getDelimiter` functions for details on the delimiter characters) followed by a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The `fileio.getu8` function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..255. This function returns the binary form of the integer in the AL register.

HLA high-level calling sequence examples:

```
fileio.getu8( fileHandle );
mov( al, u8Var );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
call fileio.getu8;
mov( al, u8Var );
```



```
fileio.getu16( Handle:dword ); @returns( "ax" );
```

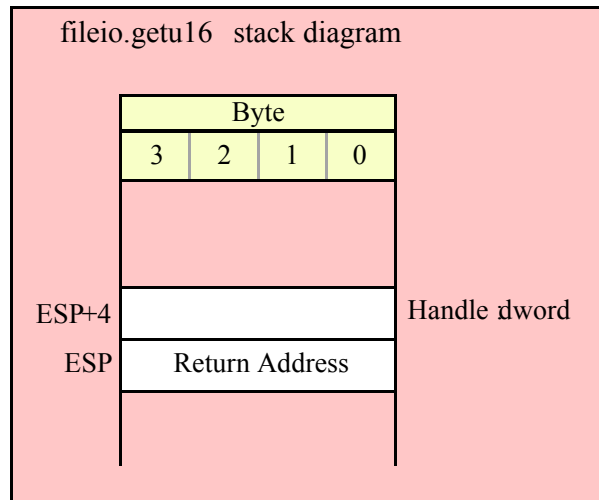
This function reads an unsigned 16-bit decimal integer in the range 0..+65535 from the file. The number may begin with any number of delimiter characters (see the `conv.setDelimiter` and `conv.getDelimiter` functions for details on the delimiter characters) followed by a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The `fileio.getu16` function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..65535. This function returns the binary form of the integer in the AX register.

HLA high-level calling sequence examples:

```
fileio.getu16( fileHandle );
mov( ax, u16Var );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
call fileio.getu16;
mov( ax, u16Var );
```



```
fileio.getu32( Handle:dword ); @returns( "eax" );
```

This function reads an unsigned 32-bit decimal integer in the range 0..+4,294,967,295 from the file. The number may begin with any number of delimiter characters (see the `conv.setDelimiter` and `conv.getDelimiter` functions for details on the delimiter characters) followed by a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The `fileio.getu32` function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..4,294,967,295. This function returns the binary form of the integer in the EAX register.

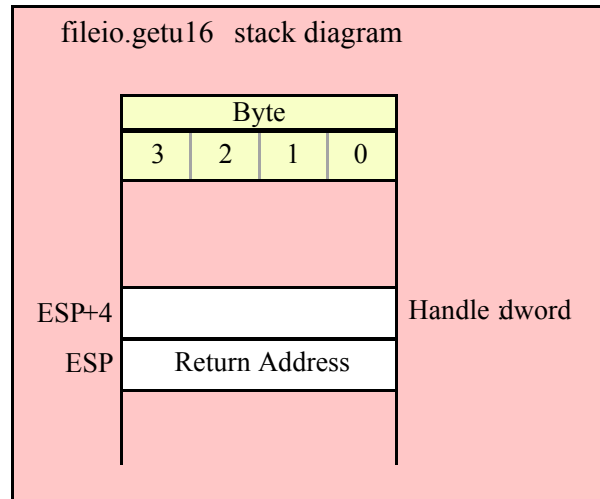
HLA high-level calling sequence examples:

```
fileio.getu32( fileHandle );
mov( eax, u32Var );
```

HLA low-level calling sequence examples:

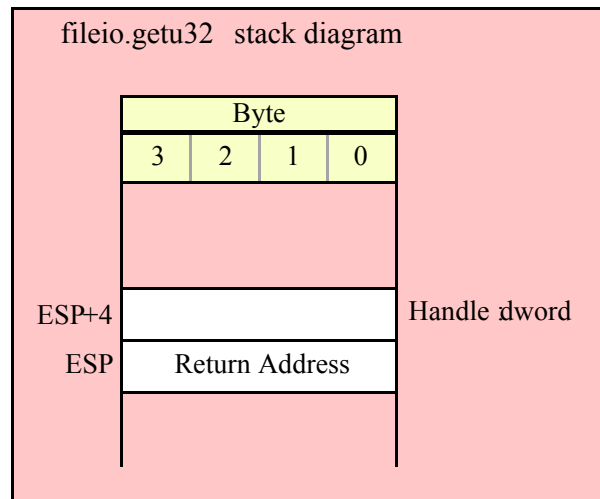
```
push( fileHandle );
call fileio.getu32;
```

```
mov( eax, u32Var );
```



```
fileio.getu64( Handle:dword );
```

This function reads an unsigned 64-bit decimal integer from the file. The number may begin with any number of delimiter characters (see the `conv.setDelimiter` and `conv.getDelimiter` functions for details on the delimiter characters) followed by a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The `fileio.getu64` function raises an appropriate exception if the input violates any of these rules or the value is outside the range  $0..2^{64}-1$ . This function returns the binary form of the integer in the the EDX:EAX registers.



```
fileio.getu128( Handle:dword; var dest:lword );
```

This function reads an unsigned 128-bit decimal integer from the file. The number may begin with any number of delimiter characters (see the `conv.setDelimiter` and `conv.getDelimiter` functions for details on the delimiter characters) followed by a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The `fileio.getu128` function raises an appropriate exception if the input violates any of these rules or the value is outside the range  $0..2^{128}-1$ . This function returns the binary form of the integer in the lword parameter you pass by reference.

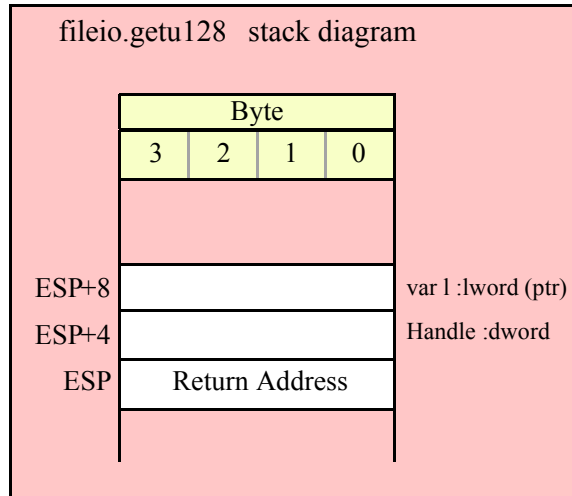
HLA high-level calling sequence examples:



```
fileio.getu64( fileHandle );
mov( edx, (type dword u64Var[4]) );
mov( eax, (type dword u64Var[0]) );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
call fileio.getu64;
mov( edx, (type dword u64Var[4]) );
mov( eax, (type dword u64Var[0]) );
```



### 15.4.5 Hexadecimal Input Routines

```
fileio.geth8( Handle:dword ); @returns( "a1" );
```

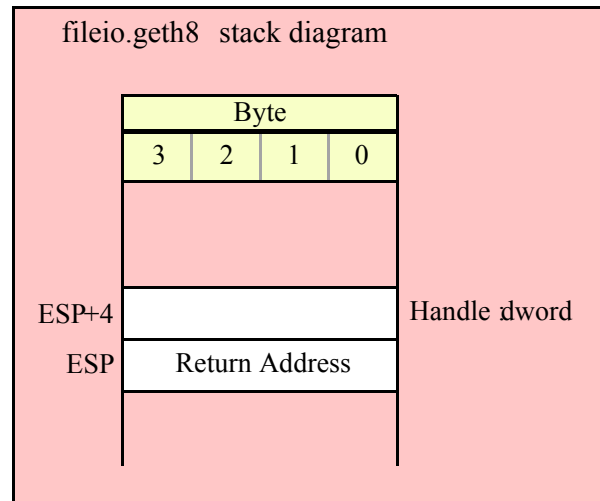
This function reads an eight-bit hexadecimal integer in the range 0..\$FF from the file. The number may begin with any number of delimiter characters (see the `conv.setDelimiter` and `conv.getDelimiter` functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "\$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The `fileio.geth` function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..\$FF. This function returns the binary form of the value in the AL register.

HLA high-level calling sequence examples:

```
fileio.geth8( fileHandle );  
mov( al, h8Var );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
call fileio.geth8;
mov( al, h8Var );
```



```
fileio.geth16( Handle:dword );    @returns( "ax" );
```

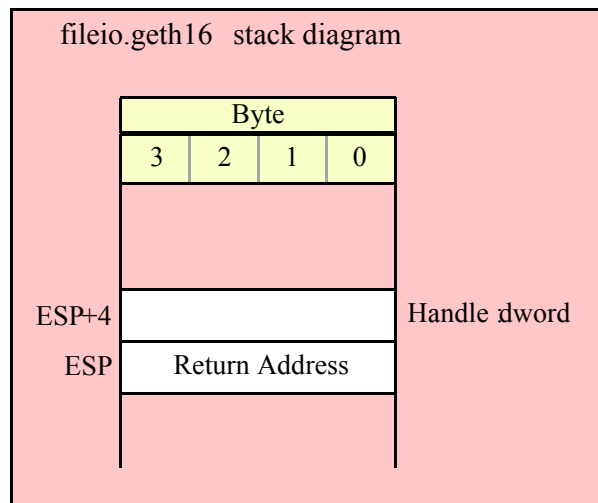
This function reads a 16-bit hexadecimal integer in the range 0..\$FFFF from the file. The number may begin with any number of delimiter characters (see the `conv.setDelimiter` and `conv.getDelimiter` functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "\$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The `fileio.geth16` function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..\$FFFF. This function returns the binary form of the value in the AX register (zero-extended into EAX).

HLA high-level calling sequence examples:

```
fileio.geth16( fileHandle );
mov( ax, h16Var );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
call fileio.geth16;
mov( ax, h16Var );
```



```
fileio.geth32( Handle:dword ); @returns( "eax" );
```

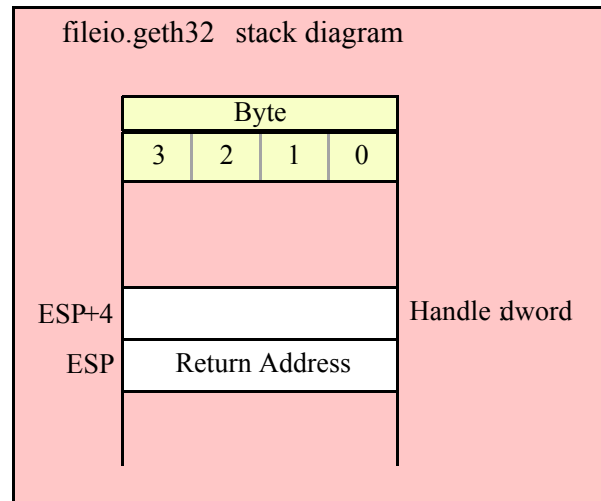
This function reads a 32-bit hexadecimal integer in the range 0..\$FFFF\_FFFF from the file. The number may begin with any number of delimiter characters (see the `conv.setDelimiter` and `conv.getDelimiter` functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "\$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The `fileio.geth32` function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..\$FFFF\_FFFF. This function returns the binary form of the value in the EAX register.

HLA high-level calling sequence examples:

```
fileio.geth32( fileHandle );
mov( eax, h32Var );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
call fileio.geth32;
mov( eax, h32Var );
```



```
fileio.geth64( Handle:dword );
```

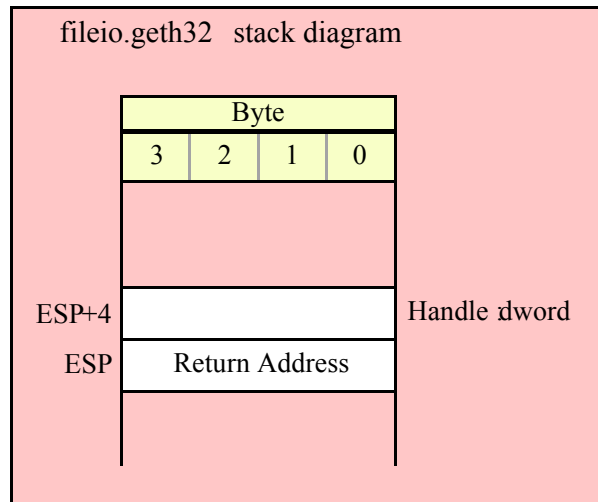
This function reads a 64-bit hexadecimal integer in the range 0..\$FFFF\_FFFF\_FFFF\_FFFF from the file. The number may begin with any number of delimiter characters (see the `conv.setDelimiter` and `conv.getDelimiter` functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "\$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The `fileio.geth64` function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..\$FFFF\_FFFF\_FFFF\_FFFF. This function returns the 64-bit result in the EDX:EAX register pair.

HLA high-level calling sequence examples:

```
fileio.geth64( fileHandle );
mov( edx, (type dword h64Var[4]) );
mov( eax, (type dword h64Var[0]) );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
call fileio.geth64;
mov( edx, (type dword h64Var[4]) );
mov( eax, (type dword h64Var[0]) );
```



```
fileio.geth128( Handle:dword; var dest:lword );
```

This function reads a 128-bit hexadecimal integer in the range zero through \$FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF from the file. The number may begin with any number of delimiter characters (see the conv.setDelimiter and conv.getDelimiter functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "\$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The fileio.geth128 function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..\$FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF. This function stores the 128-bit result into the variable you pass as a reference parameter.

HLA high-level calling sequence examples:

```
fileio.geth128( fileHandle, lwordVar );
```

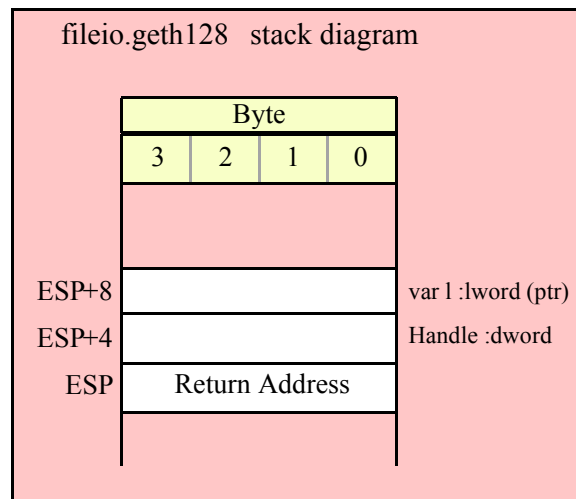
HLA low-level calling sequence examples:

```
// If lwordVar is a static variable:
```

```
push( fileHandle );
pushd( &lwordVar );
call fileio.geth128;
```

```
// If lwordVar is a not static variable
// and a 32-bit register is available:
```

```
push( fileHandle );
lea( eax, lwordVar ); // Assume EAX is available
push( eax );
call fileio.geth128;
```



### 15.4.6 Floating Point Input

```
fileio.getf( Handle:dword );
```

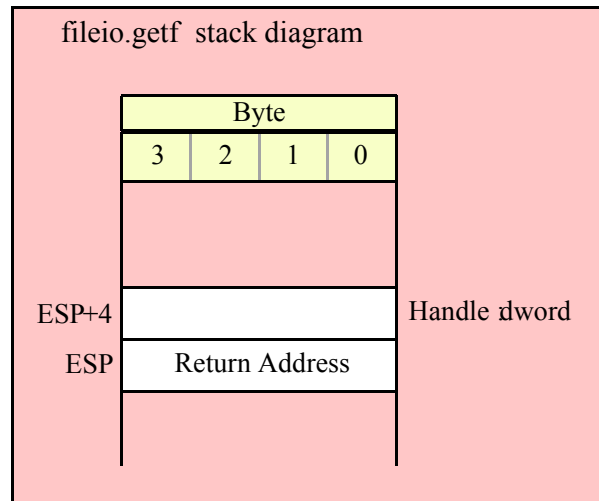
This function reads an 80-bit floating point value in either decimal or scientific from the file and leaves the result sitting on the FPU stack. The number may begin with any number of delimiter characters (see the `conv.setDelimiter` and `conv.getDelimiter` functions for details on the delimiter characters) followed by an optional minus sign and a sequence of characters that represent a floating point value. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. This function raises an appropriate exception if an error occurs.

HLA high-level calling sequence examples:

```
fileio.getf( fileHandle );
fstp( fpVar );
```

HLA low-level calling sequence examples:

```
push( fileHandle );
call fileio.getf;
fstp( fpVar );
```



## 15.4.7 Generic File Input

**fileio.get( *List\_of\_items\_to\_read* );**

This is a macro that allows you to specify a list of variable names as parameters. The fileio.get macro reads an input value for each item in the list and stores the resulting value in each corresponding variable. This macro determines the type of each variable that you pass it and emits a call to the appropriate fileio.getxxx function to read the actual value. As an example, consider the following call to filevar.get:

```
fileio.get( i32, charVar, u16, strVar );
```

The macro invocation above expands into the following:

```
push( eax );
fileio.geti32( i32 );
fileio.getc();
mov( al, charVar );
fileio.geti16();
mov( ax, u16 );
fileio.gets( strVar );
pop( eax );
```

Notice that fileio.get preserves the value in the EAX and EDX registers even though various fileio.getxxx functions use these registers. Note that fileio.get automatically handles the case where you specify EAX as an input variable and writes the value to [esp] so that it properly modifies EAX upon completion of the macro expansion.

Note that fileio.get supports eight-bit, 16-bit, 32-bit, 64-bit, and 128-bit input values. It automatically selects the appropriate input routine based on the type of the variable you specify.





## 16 The File System Module (filesys.hhf)

The filesys functions perform file system manipulations (as opposed to the fileio package that operates on files).

### 16.1 Filename and Pathname String Functions

These functions test and manipulate pathname strings. Most of them do not do any actual file system access, they simply modify a string passed to them.

The file and pathname string functions are relatively OS-neutral. As long as you avoid a few OS-specific filename features (such as Win32 drive letter prefixes and the use of backslash ['\'] characters in Unix-like OS filenames), you'll find that these functions are highly portable between various operating systems. These functions automatically convert pathname separator characters to the native OS (except for the functions that explicitly produce Win32-style or Unix-style pathnames). So you can write applications that process pathnames and automatically work with whatever OS the application is running under.

This documentation will use the standard UNIX '/' pathname separator character. Anywhere you see a "/" used in the following examples, just note that you can also use a "\" and the function will still work properly (you can even have a mixture of these two separator characters in the same pathname string and the function will accept this). Keep in mind that most filesys pathname functions will convert all '/' and '\' characters to the native directory separator character; take care if you expect to process Win32 filenames under a UNIX-like OS and you need to keep the Win32 separator characters.

For the purposes of the filesys string functions, a pathname is considered to contain up to five components: a UNC prefix, a path component, a filename component, a basename component, and an extension. These components are not necessarily unique (that is, some of them overlap one another). Not all pathnames contain all of these components. Consider the following valid pathname string:

```
fsType://computerName/SharedFolder/path1/path2/base.ext
```

This example pathname contains the following components:

```
UNC:      fsType://computerName/SharedFolder
```

```
path:     fsType://computerName/SharedFolder/path1/path2
```

```
filename:base.ext
```

```
basename:base
```

```
extensionext
```

UNC (Universal Naming Convention) names take two basic forms:

```
//computerName/sharedFolderName
```

```
fsType://computerName/sharedFolderName
```

UNC names contain an optional file system type name followed by a colon. All UNC names contain '/' followed by a computer name which is then followed by a '/' and a shared folder name.

Path components consist of everything to the left of the last '/' appearing in a path name string. Note that a UNC component is also part of a path component. Indeed, if a UNC immediately precedes a filename, then the UNC sequence is the path component (note, however, that a path component may include other subdirectory names in addition to the UNC character sequence). If there is nothing to the left of the last (i.e., only) '/' in a pathname string, then the "/" is the path component.

The filename component is everything appearing to the right of the last '/' character in the pathname string, or the whole pathname string if there is no '/' character in the pathname string.

The basename and extension components are part of the filename. If the filename contains at least one period and that period is not the first character of the filename, then everything to the left of the (last) period is the basename and everything to the right of the (last) period is the extension. If a filename contains multiple periods, then everything up to (but not including) the last period is the basename. If the filename contains only one period and it begins with that period, then the filename has no extension and the basename is equal to the filename. Likewise, if a filename contains no periods at all, it has no extension and the basename is equal to the filename.

```
procedure filesys.hasDriveLetter( pathname:string ); @returns( "@c" );
```

This function returns the Win32 drive letter if the pathname argument begins with a single alphabetic character, immediately followed by a colon (':') and the colon is not immediately followed by a pair of slashes

(indicating a UNC name). Drive letters are Win32-specific, although this function can be called on any pathname string. If a drive letter is present, this function returns the drive letter in AL, converted to uppercase, and also returns with the carry flag set. If there is no drive letter at the beginning of the name (or if it looks like a UNC name), then this function returns with EAX containing zero and the carry flag clear.

Note that the function "returns" value for this function is "@c" (that is, the carry flag) and not "AL". This allows you to use the function call within a boolean expression (e.g., in an "if" statement) and test for true/false return values.

HLA high-level calling sequence examples:

```
if( filesystem.hasDriveLetter( somePath )) then

    stdout.put( "Drive is ", (type char al), nl );
    str.delete( somePath, 0, 2 );// Delete the drive letter

endif;
```

HLA low-level calling sequence example:

```
push( somePath );
call filesystem.hasDriveLetter;
jnc noDriveLetter;

    push( somePath );
    pushd( 0 );
    pushd( 2 );
    call str.delete;

noDriveLetter:
```

**procedure filesystem.hasExtension( pathname:string ); @returns( "@c" );**

This function returns true if the filename component of the pathname argumen contains an extension. An extension is the last part of a pathname following the last period in the filename. Note that if a filename begins with a period and that is the only period in the filename, then the following characters are not an extension (and the extension is the empty string for such a name). Note that periods found in the path to the filename are not considered when this function searches for the extension. Extensions only belong to filename components, not to path components.

Examples:

```
filesystem.hasExtension( "/path/file.ext" );// true, extension = "ext"
filesystem.hasExtension( "file" );// false
filesystem.hasExtension( ".ext" );// false
filesystem.hasExtension( "file.ext" );// true, extension = "ext"
filesystem.hasExtension( "file.abc.ext" );// true, extension = "ext"
filesystem.hasExtension( "..ext" );// true, extension = "ext"
filesystem.hasExtension( "path.ext/file" );// false
```

The true/false result is returned both in the carry flag and in the EAX/AX/AL register. The carry flag is set if the argument has an extension, it is clear otherwise. Similarly, true (1) is returned in EAX/AX/AL if the argument has an extension, false (0) is returned otherwise.

HLA high-level calling sequence examples:

```
filesystem.hasExtension( somePath );
mov( al, somePathHasExtension );
if( @c ) then

    << do something if somePath has an extension >>
```

```
endif;
```

HLA low-level calling sequence example:

```
push( somePath );
call filesystems.hasExtension;
jnc noExtension;

    << Do something if somePath has an extension >>

noExtension:
```

```
procedure filesystems.hasUncName( pathname:string ); @returns( "@c" );
```

This function tests the pathname argument to see if it begins with a UNC (universal naming convention) pathname prefix. UNC prefixes take one of two forms (as far as this code is concerned):

```
//computername/sharedfolder/<path>
<type>://computername/sharedfolder/<path>
```

<type> can be any string of filename-compatible characters (length one or greater), such as 'file', 'smb', and so on. <path> may be any OS-compatible pathname of length zero or greater (up to the maximum length supported by the native OS). Portable code should not allow the pathname string (including the UNC) to exceed about 250 characters.

This function returns true or false in the carry flag indicating whether a UNC, if present, is syntactically correct. That is, this function returns carry clear if there is something that looks like a UNC but is syntactically illegal. Note that a pathname, without an explicit "//computername/sharedfolder/" prefix is still a syntactically correct pathname as far as this function is concerned. That is, this function returns true for pathnames like "name" or "/path/name" even though an explicit UNC item is not present.

This function returns information about the UNC prefix in the EAX register. If this function returns with EAX equal to zero and the carry set, then there is no UNC present in the pathname. If this function returns with EAX containing a value other than zero (and the carry flag set), then a UNC is present and EAX contains an offset into the string that is the start of the pathname just beyond the end of the UNC sequence (i.e., beyond the '/' or '\' that marks the end of the UNC name).

Note: on failure (carry = 0), EAX will be returned containing zero.

HLA high-level calling sequence examples:

```
if( filesystems.hasUncName( somePath ) && eax <> 0 ) then

    str.delete( somePath, 0, eax );// Delete the UNC prefix

endif;
```

HLA low-level calling sequence example:

```
push( somePath );
call filesystems.hasUncName;
jnc noUNC;
test( eax, eax );
jz noUNC;

    push( somePath );
    pushd( 0 );
    push( eax );
    call str.delete;
```

noUNC:

```
procedure filesystems.hasPath( pathname:string ); @returns( "@c" );
```

This function returns true if the pathname/filename argument contains a path component. Specifically, this function returns true if the pathname string contains any directory separator characters ('/' or '\'). True is returned in the carry flag (set) and in the EAX/AX/AL register (1). False is carry = 0 or the EAX/AX/AL register equals 0. Note that UNC prefixes immediately before a filename are considered 'paths' and this function will return true if a UNC is present.

Examples:

```
filesystems.hasPath( "/path/file.ext" );// true
filesystems.hasPath( "file" ); // false
filesystems.hasPath( ".ext" ); // false
filesystems.hasPath( "file.ext" );// false
filesystems.hasPath( "file.abc.ext" );// false
filesystems.hasPath( "//machine/folder/file" );// true
filesystems.hasPath( "/" ); // true
```

HLA high-level calling sequence examples:

```
filesystems.hasPath( somePath );
mov( al, somePathHasAPathComponent );
if( @c ) then

    << do something if somePath has a path component>>

endif;
```

HLA low-level calling sequence example:

```
push( somePath );
call filesystems.hasPath;
jnc noPath;

    << Do something if somePath has a path component>>

noPath:
```

```
procedure filesystems.a_extractBase( pathname:string ); @returns( "c" );
```

This function extracts and returns the base component of a filename. This function allocates storage for the returned basename on the heap and returns a pointer to that string in the EAX register. If a basename exists, this function returns true in the carry flag (set). If no basename exists (e.g., when the filename ends with '/' so it contains no filename component or if the pathname argument is the empty string), then this function returns a pointer to an empty string allocated on the heap in EAX and it returns with the carry flag clear. It is the caller's responsibility to free the storage associated with the string when the caller is done using that string.

Note that this function's "returns" value is "@c", not EAX. This allows you to use this function in an HLA boolean expression (e.g., in an "if" statement) to test whether the basename is actually valid (that is, it's not an empty string). Don't forget that you still have to free the storage associated with the string, even if it is the empty string.

HLA high-level calling sequence examples:

```
filesystems.a_extractBase( somePath );
```

```

mov( eax, basePtr );
if( @c ) then

    <<do something with the basename pointed at by basePtr>>

endif;
str.free( basePtr );

```

HLA low-level calling sequence example:

```

push( somePath );
call filesys.a_extractBase;
mov( eax, basePtr );
jnc noPath;

    << Do something if somePath has a path component>>

noPath:
str.free( basePtr );

```

```

procedure filesys.extractBase( pathname:string; base:string );
    @returns( "c" );

```

This function extracts and returns the base component of a filename. It extracts the base filename from the *pathname* argument and stores the result into the string storage pointed at by the *base* argument. The *base* argument must point at allocated storage sufficient to hold the base name string or this function will raise an exception. This function returns with the carry flag set if it finds (and copies) a basename component of the *pathname*. It returns with the carry flag clear if there is no base name component (which implies that *pathname* ends with a '/' character or is the empty string).

HLA high-level calling sequence examples:

```

filesys.extractBase( somePath, baseName );
if( @c ) then

    <<do something with the basename held in baseName>>

endif;

```

HLA low-level calling sequence example:

```

push( somePath );
push( baseName );
call filesys.extractBase;
jnc noPath;

    << Do something if somePath has a path component>>

noPath:

```

```
procedure filesystem.a_extractExt( pathname:string ); @returns( "c" );
```

This function extracts and returns the extension component of a filename. This function allocates storage for the returned extension on the heap and returns a pointer to that string in the EAX register. If an extension exists, this function returns true in the carry flag (set). If no extension exists (e.g., when the filename component contains no periods), then this function returns a pointer to an empty string allocated on the heap in EAX and it returns with the carry flag clear. It is the caller's responsibility to free the storage associated with the string when the caller is done using that string.

Note that this function's "returns" value is "@c", not EAX. This allows you to use this function in an HLA boolean expression (e.g., in an "if" statement) to test whether the extension is actually valid (that is, it's not an empty string). Don't forget that you still have to free the storage associated with the string, even if it is the empty string.

HLA high-level calling sequence examples:

```
filesystem.a_extractExt( somePath );
mov( eax, extPtr );
if( @c ) then

    <<do something with the extension pointed at by extPtr>>

endif;
str.free( extPtr );
```

HLA low-level calling sequence example:

```
push( somePath );
call filesystem.a_extractExt;
mov( eax, extPtr );
jnc noPath;

    << Do something if somePath has an extension component>>

noPath:
str.free( extPtr );
```

```
procedure filesystem.extractExt( pathname:string; ext:string ); @returns( "c" );
```

This function extracts and returns the extension component of a filename. It extracts the extension from the *pathname* argument and stores the result into the string storage pointed at by the *ext* argument. The *ext* argument must point at allocated storage sufficient to hold the extension string or this function will raise an exception. This function returns with the carry flag set if it finds (and copies) an extension component of the pathname. It returns with the carry flag clear if there is no extension component (which implies that filename component contains no periods or is the empty string).

HLA high-level calling sequence examples:

```
filesystem.extractExt( somePath, extName );
if( @c ) then

    <<do something with the extension held in extName>>

endif;
```

HLA low-level calling sequence example:

```
push( somePath );
```

```

push( extName );
call filesys.extractExt;
jnc noPath;

    << Do something if somePath has an extension component>>

noPath:

```

**procedure filesys.a\_extractFilename( pathname:string ); @returns( "c" );**

This function extracts and returns the filename component of a pathname. This function allocates storage for the returned filename on the heap and returns a pointer to that string in the EAX register. If a filename component exists, this function returns true in the carry flag (set). If no filename exists (e.g., when the pathname ends with a '/' or is the empty string), then this function returns a pointer to an empty string allocated on the heap in EAX and it returns with the carry flag clear. It is the caller's responsibility to free the storage associated with the string when the caller is done using that string.

Note that this function's "returns" value is "@c", not EAX. This allows you to use this function in an HLA boolean expression (e.g., in an "if" statement) to test whether the filename is actually valid (that is, it's not an empty string). Don't forget that you still have to free the storage associated with the string, even if it is the empty string.

HLA high-level calling sequence examples:

```

filesys.a_extractFilename( somePath );
mov( eax, fnPtr );
if( @c ) then

    <<do something with the filename pointed at by fnPtr>>

endif;
str.free( fnPtr );

```

HLA low-level calling sequence example:

```

push( somePath );
call filesys.a_extractFilename;
mov( eax, fnPtr );
jnc noPath;

    << Do something if somePath has a filename component>>

noPath:
str.free( fnPtr );

```

**procedure filesys.extractFilename( pathname:string; filename:string );**  
**@returns( "c" );**

This function extracts and returns the filename component of a pathname. It extracts the filename from the *pathname* argument and stores the result into the string storage pointed at by the *filename* argument. The *filename* argument must point at allocated storage sufficient to hold the filename string or this function will raise an exception. This function returns with the carry flag set if it finds (and copies) a filename component of the pathname. It returns with the carry flag clear if there is no filename component (which implies that pathname component ends with a '/' or is the empty string).

HLA high-level calling sequence examples:

```

filesys.extractFilename( somePath, filename );
if( @c ) then

    <<do something with the string held in filename>>

endif;

```

HLA low-level calling sequence example:

```

push( somePath );
push( filename );
call filesys.extractFilename;
jnc noPath;

    << Do something if somePath has a filename component>>

noPath:

```

**procedure filesys.a\_extractPath( pathname:string ); @returns( "c" );**

This function extracts and returns the path component of a pathname string. This function allocates storage for the returned path on the heap and returns a pointer to that string in the EAX register. If a path component exists, this function returns true in the carry flag (set). If no path component exists (e.g., when the pathname argument contains no '/' characters or is the empty string), then this function returns a pointer to an empty string allocated on the heap in EAX and it returns with the carry flag clear. It is the caller's responsibility to free the storage associated with the string when the caller is done using that string.

Note that this function's "returns" value is "@c", not EAX. This allows you to use this function in an HLA boolean expression (e.g., in an "if" statement) to test whether the path is actually valid (that is, it's not an empty string). Don't forget that you still have to free the storage associated with the string, even if it is the empty string.

HLA high-level calling sequence examples:

```

filesys.a_extractPath( somePath );
mov( eax, pathPtr );
if( @c ) then

    <<do something with the path pointed at by pathPtr>>

endif;
str.free( pathPtr );

```

HLA low-level calling sequence example:

```

push( somePath );
call filesys.a_extractPath;
mov( eax, pathPtr );
jnc noPath;

    << Do something if somePath has a path component>>

noPath:
str.free( pathPtr );

```



```

procedure filesystems.extractPath( pathname:string; path:string );
    @returns( "c" );

```

This function extracts and returns the path component of a pathname string. It extracts the path from the *pathname* argument and stores the result into the string storage pointed at by the *path* argument. The *path* argument must point at allocated storage sufficient to hold the string or this function will raise an exception. This function returns with the carry flag set if it finds (and copies) a path component of the pathname. It returns with the carry flag clear if there is no path component (which implies that pathname component contains no '/' characters or is the empty string).

Note that the path string that this function returns will include any UNC prefixes and any Win32 drive letters that are present in the original pathname. If you need a path result that doesn't contain any drive letter prefixes, call *filesystems.hasDriveLetter* on the result and delete the first two character of the string if there is a drive letter present. If you need a string without a UNC prefix present, then call *filesystems.hasUncName* on the path result and delete the first EAX characters from the path string if *filesystems.hasUncName* reports that a UNC name is present.

HLA high-level calling sequence examples:

```

filesystems.extractPath( somePath, pathComponent );
if( @c ) then

    <<do something with the string held in pathComponent >>

endif;

```

HLA low-level calling sequence example:

```

push( somePath );
push( pathComponent );
call filesystems.extractPath;
jnc noPath;

    << Do something if somePath has a path component>>

noPath:

```

```

procedure filesystems.a_joinPaths( leftPath:string; rightPath:string );
    @returns( "eax" );

```

This function concatenates two path strings, adding a directory separator character between them (if necessary). Because of the wide variety of special cases that can occur when concatenating two paths, this function is fairly complex. The operation is described in the following table.

If leftPath...	If rightPath...	Then the resulting path string...
Is empty	Is empty	Is empty.
Ends with '/'	Does not begin with '/'	Is just the concatenation of the two strings.

Does not end with '/'	Begins with '/'	Is just the concatenation of the two strings.
Ends with '/'	Begins with '/'	Is the concatenation of the two strings with one of the '/' characters removed.
Does not end with '/'	Does not begin with '/'	Is the concatenation of the leftPath with '/' and then the rightPath string.

Note that, unlike many of the other *filesys* file/path string functions, this function does not return a failure/success status in the carry flag. This function always succeeds (or it raises an exception if an exceptional condition exists).

HLA high-level calling sequence examples:

```
filesys.a_joinPaths( leftPath, RightPath );
mov( eax, pathPtr );
```

<<do something with the path pointed at by pathPtr>>

```
str.free( pathPtr );
```

HLA low-level calling sequence example:

```
push( leftPath );
push( rightPath
call filesys.a_joinPaths;
mov( eax, pathPtr );
```

<< Do something with the string pointed at by pathPtr>>

```
str.free( pathPtr );
```

```
procedure filesys.joinPaths
(
    leftPath:string;
    rightPath:string;
    joinedPath:string
);
```

This function combines the *leftPath* and *rightPath* strings to form a *joinedPath* string. For the exact details on the concatenation operation, please see the table appearing in the description of the *filesys.a\_joinPaths* function. This function will raise an exception if the string storage pointed at by *joinedPaths* is not large enough to hold the result. Note that the calculation for string overflow is computed as the sum of the lengths of *leftPath* and *rightPath* plus one, even though (in some cases) the required length might need only be the sum of the lengths of the *leftPath* and *rightPath* strings. Therefore, you should ensure that the storage allocated for the *joinedPath* string is at least one character larger than the sum of the two substrings or this function may raise an exception, even if *joinedPath* turns out to be large enough to hold the actual result.

HLA high-level calling sequence examples:

```
fileSYS.joinPaths( leftPath, rightPath, newPath );

<<do something with the string held in newPath >>
```

HLA low-level calling sequence example:

```
push( leftPath );
push( rightPath );
push( newPath
call fileSYS.joinPaths;

<< Do something with newPath>>
```

**procedure fileSYS.a\_normalize( pathname:string ); @returns( "c" );**

This function normalizes the pathname passed as the argument and returns a pointer to the normalized pathname in EAX. This function allocates storage for the normalized pathname on the heap; it is the caller's responsibility to free that storage when the application is done using the string.

A normalized pathname is one that has all the directory separators converted to the native format, has all path components of the form `"/"` deleted from the path string, and has all path components of the form `"path/.."` deleted from the path string. Note, however, that if `"/"` appears at the beginning of a path string, or `"/"` appears immediately after a UNC path sequence, then the normalized result still contains the `"/"`. Likewise, if there are multiple `"/"` sequences within a path string and deleting the previous paths would delete a UNC component or would attempt to delete a path sequence before any appearing in the path string, then this function leaves the `"/"` component present (e.g., `"path/../../name"` produces the string `"/name"`).

This function returns the carry flag set if it was able to produce a correct normalized string. This function returns with the carry flag clear if the *pathname* argument contained an unparseable UNC name prefix or other syntax error (in which case the function ignores the UNC prefix and treats the UNC like any other path sequence).

HLA high-level calling sequence examples:

```
fileSYS.a_normalize( somePath );
mov( eax, pathPtr );
if( @c ) then

    <<do something with the path pointed at by pathPtr>>

endif;
str.free( pathPtr );
```

HLA low-level calling sequence example:

```
push( somePath );
call fileSYS.a_extractPath;
mov( eax, pathPtr );
jnc noPath;

<< Do something if somePath has a path component>>

noPath:
str.free( pathPtr );
```

```

procedure filesystems.normalize1( pathname:string );
    @returns( "c" );

```

This function normalizes, in place, the *pathname* string passed as an argument. On returns, the *pathname* string contains the normalized result of the original string passed into this function. Note that normalized strings are always the same length or shorter than the original string, so there is no chance of string overflow occurring when normalizing a path string. For details on the normalization process, see the description of the *filesystems.a\_normalize* function.

This function returns with the carry flag set if it successfully normalizes the *pathname* argument and there are no UNC parse errors or other problems. If this function cannot parse a string that looks like it has a UNC prefix, it will treat *pathname* as though it has no UNC prefix, normalize that, and return with the carry flag clear.

Because this function normalizes the string in place, the *pathname* argument must point at string data in writeable memory or else this function will raise an exception.

HLA high-level calling sequence examples:

```

filesystems.normalize1( somePath );
if( @c ) then

    <<do something with the string held in somePath >>

endif;

```

HLA low-level calling sequence example:

```

push( somePath );
call filesystems.normalize1;
jnc noPath;

    << Do something with somePath>>

noPath:

```

```

procedure filesystems.normalize2( pathname:string; path:string );
    @returns( "c" );

```

This function normalizes the *pathname* string passed as an argument and stores the normalized result into the string object pointed at by *path*. The *path* object must have at least as much space allocated for it as the length of the *pathname* argument or this function will raise an exception; this exception will be raised even if the actual normalized string would be shorter than the length of *pathname* and *path* is actually large enough to hold the actual normalized string. The test for string overflow takes place before the normalization operation begins because this function first copies *pathname* to *path* and then performs the normalization operation on *path*. For details on the normalization process, see the description of the *filesystems.a\_normalize* function.

This function returns with the carry flag set if it successfully normalizes the *pathname* argument and there are no UNC parse errors or other problems. If this function cannot parse a string that looks like it has a UNC prefix, it will treat *pathname* as though it has no UNC prefix, normalize that, and return with the carry flag clear.

HLA high-level calling sequence examples:

```

filesystems.extractPath( somePath, pathComponent );
if( @c ) then

    <<do something with the string held in pathComponent >>

```

```
endif;
```

HLA low-level calling sequence example:

```
push( somePath );
push( pathComponent );
call filesys.extractPath;
jnc noPath;

    << Do something if somePath has a path component>>

noPath:
```

```
procedure filesys.a_toUnixPath( pathname:string ); @returns( "eax" );
```

This function converts a string to the Unix format. This entails converting all ‘\’ characters to ‘/’ characters in the pathname string. Note that this function does not process any drive letter prefixes. If a drive letter prefix is present in the pathname argument, this function returns that drive letter prefix in the result string.

This function returns a pointer to the converted string, which is allocated on the heap, in the EAX register. It is the caller’s responsibility to free the storage associated with this string when the caller is done using the string data.

HLA high-level calling sequence examples:

```
filesys.a_toUnixPath( somePath );
mov( eax, unixPathPtr );

    <<do something with the path pointed at by unixPathPtr >>

str.free( unixPathPtr );
```

HLA low-level calling sequence example:

```
push( somePath );
call filesys.a_toUnixPath;
mov( eax, unixPathPtr );

    << Do something with unixPathPtr>>

str.free( unixPathPtr );
```

```
procedure filesys.toUnixPath1( pathname:string );
```

This function converts the string argument to UNIX format by replacing all ‘\’ characters with ‘/’ characters. Note that this function does not process any drive letter prefixes. If a drive letter prefix is present in the pathname argument, this function returns that drive letter prefix in the result string. Converted strings are always the same length as the original string, so there is no chance of string overflow occurring when converting a path string to UNIX format.

Because this function normalizes the string in place, the *pathname* argument must point at string data in writeable memory or else this function will raise an exception.

HLA high-level calling sequence examples:

```

filesys.toUnixPath1( somePath );

<<do something with the string held in somePath >>

```

HLA low-level calling sequence example:

```

push( somePath );
call filesys.toUnixPath1;

<< Do something with somePath>>

```

### **procedure filesys.toUnixPath2( pathname:string; unixPath:string );**

This function converts the *pathname* string argument to UNIX format by replacing all ‘\’ characters with ‘/’ characters. It stores the resulting string into *unixPath*. Note that this function does not process any drive letter prefixes. If a drive letter prefix is present in the *pathname* argument, this function returns that drive letter prefix in the result string. Converted strings are always the same length as the original string, so the storage pointed at by the *unixPath* argument must be able to hold at least as many characters as the current length of the *pathname* argument or this function will raise an exception.

HLA high-level calling sequence examples:

```

filesys.toUnixPath2( somePath, unixPath );

<<do something with the string held in unixPath>>

```

HLA low-level calling sequence example:

```

push( somePath );
push( unixPath );
call filesys.toUnixPath2;

<< Do something with unixPath>>

```

### **procedure filesys.a\_toWin32Path( pathname:string ); @returns( "eax" );**

This function converts a *pathname* string to the Windows format. This entails converting all ‘/’ characters to ‘\’ characters in the *pathname* string.

This function returns a pointer to the converted string, which is allocated on the heap, in the EAX register. It is the caller’s responsibility to free the storage associated with this string when the caller is done using the string data.

HLA high-level calling sequence examples:

```

filesys.a_toWin32Path( somePath );
mov( eax, win32PathPtr );

<<do something with the path pointed at by win32PathPtr >>

```

```
str.free( win32PathPtr );
```

HLA low-level calling sequence example:

```
push( somePath );
call filesys.a_toWin32Path;
mov( eax, win32PathPtr );

    << Do something with win32PathPtr >>

str.free( win32PathPtr );
```

**procedure filesys.toWin32Path1( pathname:string );**

This function converts the *pathname* string argument to Windows format by replacing all ‘/’ characters with ‘\’ characters. Converted strings are always the same length as the original string, so there is no chance of string overflow occurring when converting a path string to UNIX format.

Because this function normalizes the string in place, the *pathname* argument must point at string data in writeable memory or else this function will raise an exception.

HLA high-level calling sequence examples:

```
filesys.toWin32Path1( somePath );

    <<do something with the string held in somePath >>
```

HLA low-level calling sequence example:

```
push( somePath );
call filesys.toWin32Path1;

    << Do something with somePath>>
```

**procedure filesys.toWin32Path2( pathname:string; windowsPath:string );**

This function converts the *pathname* string argument to Windows format by replacing all ‘/’ characters with ‘\’ characters. It stores the resulting string into *windowsPath*. Converted strings are always the same length as the original string, so the storage pointed at by the *windowsPath* argument must be able to hold at least as many characters as the current length of the *pathname* argument or this function will raise an exception.

HLA high-level calling sequence examples:

```
filesys.toWin32Path2( somePath, windowsPath );

    <<do something with the string held in windowsPath >>
```

HLA low-level calling sequence example:

```

push( somePath );
push( windowsPath );
call filesys.toWin32Path2;

<< Do something with windowsPath >>

```

```

procedure filesys.a_toNativePath( pathname:string );
procedure filesys.toNativePath1( pathname:string );
procedure filesys.toNativePath2( pathname:string; windowsPath:string );

```

These functions are synonyms for either the *toUnix* functions or the *toWin32* functions, depending upon the operating system under which you're compiling them. That is, under Windows, these functions are synonyms for the *filesys.a\_toWin32Path*, *filesys.toWin32Path1*, and *filesys.toWin32Path2* functions. Under other Oses, these functions are synonyms for the *filesys.a\_toUnixPath*, *filesys.toUnixPath1*, and *filesys.toUnixPath2* functions.

Functionally, this procedures convert the *pathname* passed as an argument to the native OS pathname format. Note that these functions ignore drive letters if they are converting pathnames to UNIX format (that is, the drive letters will still be present in the converted string).

```

procedure filesys.a_getFullPathName( partialPath:string ); @returns( "eax" );

```

This function takes the *partialPath* passed as a parameter and converts it to a full, absolute, pathname. If *partialPath* begins with a '/' character, this function simply returns *partialPath*'s value as its result. If *partialPath* does not begin with a '/' character, then this function determines the working directory path and concatenates (via *filesys.joinPaths*) *partialPath* to the end of the current working directory and returns that full path string.

This function returns a pointer to the full pathname string, which is allocated on the heap, in the EAX register. It is the caller's responsibility to free the storage associated with this string when the caller is done using the string data.

HLA high-level calling sequence examples:

```

filesys.a_getFullPath( somePath );
mov( eax, fullPathPtr );

<<do something with the path pointed at by fullPathPtr>>

str.free( fullPathPtr );

```

HLA low-level calling sequence example:

```

push( somePath );
call filesys.a_getFullPath;
mov( eax, fullPathPtr );

<< Do something with fullPathPtr >>

str.free( fullPathPtr );

```

```

procedure filesys.getFullPath( partialPath:string; resultPath:string );

```

This function takes the *partialPath* passed as a parameter and converts it to a full, absolute, pathname. If *partialPath* begins with a '/' character, this function simply returns *partialPath*'s value as its result. If *partialPath* does not begin with a '/' character, then this function determines the working directory path and concatenates (via *filesys.joinPaths*) *partialPath* to the end of the current working directory and returns that full path string. In any case, the resulting fully qualified pathname is stored into the *resultPath* string (raising an exception if *resultPath*'s allocation is insufficient to hold the string).



HLA high-level calling sequence examples:

```
fileSYS.getFullPath( somePath, fullPath );

<<do something with the string held in fullPath>>
```

HLA low-level calling sequence example:

```
push( somePath );
push( fullPath );
call fileSYS.getFullPath;

<< Do something with fullPath>>
```

## 16.2 Directory and File Predicates

These functions test some condition about a file or directory and return true or false in the EAX register.

**procedure fileSYS.exists( pathname:string ); @returns( "eax" );**

This function returns true if the file exists and the application can open the file (at least for reading). It returns false in EAX if the file does not exist, is inaccessible, or there is some other error that occurs when attempting to open the file.

HLA high-level calling sequence examples:

```
fileSYS.exists( "someFilename" );
mov( al, someFilenameExists );// someFileNameExists:boolean;
```

HLA low-level calling sequence examples:

```
someFilename:string := "someFileName";
.
.
.
push( someFileName );
call fileSYS.exists;
mov( al, someFilenameExists );
```

**procedure fileSYS.isFile( FileName:string ); @returns( "eax" );**

This function returns true if the file exists and is actually a file (rather than a directory) and the application can open the file (at least for reading). It returns false in EAX if the file does not exist, exists but is a directory, is inaccessible, or there is some other error that occurs when attempting to open the file.

HLA high-level calling sequence examples:

```
fileSYS.isFile( "someFilename" );
mov( al, someFilenameExists );// someFileNameExists:boolean;
```

HLA low-level calling sequence examples:

```
someFilename:string := "someFileName";
.
.
.
```

```

push( someFileName );
call filesys.isFile;
mov( al, someFilenameExists );

```

```
procedure filesys.isDir( FileName:string ); @returns( "eax" );
```

This function returns true if the file exists and is a directory (rather than a regular file). It returns false in EAX if the file does not exist, exists but is not a directory, is inaccessible, or there is some other error that occurs when attempting to open the file.

HLA high-level calling sequence examples:

```

filesys.isDir( "someDirName" );
mov( al, someDirNameExists );// someDirNameExists:boolean;

```

HLA low-level calling sequence examples:

```

someDirName:string := "someDirName";
.
.
.
push( someDirName );
call filesys.isDir;
mov( al, someDirNameExists );

```

## 16.3 File Information Functions

These functions return some information about the file.

```
procedure filesys.size( Handle:dword ); @returns( "edx:eax" );
procedure filesys.size( filename:string ); @returns( "edx:eax" );
```

These (overloaded) functions return the current size of a file. They return the size in the EDX:EAX register pair. There are two versions of this function – one that accepts a string parameter and one that accepts a dword parameter. The first (dword) form expects you to pass it an open file handle; the second (string) form expects you to pass it a string containing the filename of the file whose size you want to compute.

Note: HLA uses macros to implement overloading. If you really must make a low-level call to one of these functions (or if you need to take the address of one of these functions), then you will need to refer to the actual procedure names: *filesys.\_sizeh\_* is the name of the function expecting a file handle, *filesys.\_sizen\_* is the name of the function expecting a filename string parameter.

HLA high-level calling sequence examples:

```

filesys.size( "someFileName" );
mov( eax, sizeOfSomeFileName );
filesys._sizen_( "someOtherFileName" );
mov( eax, sizeOfSomeOtherFile );

```

```

fileio.open( "anotherFile", fileio.r );
mov( eax, fileHandle );
filesys.size( fileHandle );
mov( eax, sizeOfAnotherFile );

```

```

fileio.open( "anotherFile2", fileio.r );
mov( eax, fileHandle2 );
filesys._sizeh_( fileHandle2 );
mov( eax, sizeOfAnotherFile2 );

```

HLA low-level calling sequence examples:

```

someFileName:string := "someFileName";
anotherFile:string := "anotherFile";
.
.
.
push( someFileName );
callfilesys._sizen_;
mov( eax, sizeOfSomeOtherFile );
.
.
.
push( anotherFile );
pushd( fileio.r );
call fileio.open;
mov( eax, fileHandle );
push( eax );
call filesys._sizeh_;
mov( eax, sizeOfAnotherFile2 );

```

## 16.4 Directory and File Manipulation Functions

**procedure filesys.delete( filename:string ); @returns( "eax" );**

This function deletes the specified file. Obviously, use this function with care. If this function fails, it raises the *ex.CannotRemoveFile* exception. Note that this function will only delete regular files. It will fail, and raise an exception if you attempt to delete a directory.

HLA high-level calling sequence examples:

```

filesys.delete( "someFileName" );

```

HLA low-level calling sequence examples:

```

someFileName:string := "someFileName";
.
.
.
push( someFileName );
call filesys.delete;

```

**procedure filesys.mkdir( dirname:string ); @returns( "eax" );**

This function creates a directory using the pathname you supply as a parameter. It returns the error status in EAX. If this function fails, it raises the *ex.CannotCreateDir* exception.

HLA high-level calling sequence examples:

```

filesys.mkdir( "newDirName" );

```

HLA low-level calling sequence examples:

```

newDirName:string := "newDirName";
.

```

```

.
.
push( newDirName);
call filesys.mkdir;

```

**procedure filesys.cd( dirname:string );**

This function sets the current working directory to the filename you pass as a parameter. If this function fails, it raises the *ex.CDFailed* exception.

HLA high-level calling sequence examples:

```
filesys.cd( "newWorkingDirectory" );
```

HLA low-level calling sequence examples:

```

newWorkingDirectory:string := "newWorkingDirectory";
.
.
.
push( newWorkingDirectory);
call filesys.cd;

```

**procedure filesys.gwd( dest:string );**

This function returns a string containing the current working directory's pathname in the string you pass as a parameter. The string must have storage allocated for it and it must be large enough to hold the pathname or HLA will raise a string overflow exception.

HLA high-level calling sequence examples:

```
filesys.gwd( allocatedStringVar );
```

HLA low-level calling sequence examples:

```

push( allocatedStringVar);
call filesys.gwd;

```

**procedure filesys.rename( fromPath:string; toPath:string );**

This function renames one file to another. The *fromPath* parameter specifies the file to rename, the *toPath* parameter specifies the new name for the file. If the rename operation is unsuccessful, this function raises the *ex.CannotRenameFile* exception.

HLA high-level calling sequence examples:

```
filesys.rename( "oldName", "newName" );
```

HLA low-level calling sequence examples:

```

oldName:string := "oldName";
newName:string := "newName";
.
.

```

```

    .
    push( oldName );
    push( newName );
    call filesys.rename;

```

#### **procedure filesys.rmdir( directory:string );**

This function deletes the specified directory. The directory must be empty before you attempt to delete it or this function will raise an exception. If this function fails, it raises the *ex.CannotRemoveDir* exception. Note that this function will only delete directories. It will fail, and raise an exception if you attempt to delete a regular file.

HLA high-level calling sequence examples:

```
filesys.rmdir( "dirToRemove" );
```

HLA low-level calling sequence examples:

```

dirToRemove:string := "dirToRemove";
.
.
.
push( dirToRemove );
call filesys.rmdir;

```

#### **iterator filesys.fileWithSuffix( directory:string; suffix:string );**

This iterator, used within a foreach loop, will repeat once for each file in the directory specified by the *directory* parameter that ends with the string specified by *suffix*. On each iteration of the foreach loop, this function will provide a pointer to the full filename in the EAX register. Note that this function only iterates on regular files, it will not iterate (nor return the string name) of any directory entries. The *filesys.fileWithSuffix* function will allocate storage for the filename string on the heap and return the pointer to this string in EAX. It is the foreach loop body's responsibility to free up that storage when it is done with the string.

HLA high-level calling sequence examples:

```

foreach filesys.fileWithSuffix( "dirToSearch", ".hla" ) do

    // At this point, EAX points at a filename ending with ".hla"

    mov( eax, filename );// filename:string

    // Do something with that string...

    // When we're done with the string, free it.

    str.free( filename );

endfor;

```

#### **iterator filesys.fileIn( directory:string);**

This iterator, used within a foreach loop, will repeat once for each file in the directory specified by the *directory* parameter. On each iteration of the foreach loop, this function will provide a pointer to the full filename in the EAX register. Note that this function only iterates on regular files, it will not iterate (nor return the string name) of any directory entries. The *filesys.fileIn* function will allocate storage for the filename string on the heap and return the pointer to this string in EAX. It is the foreach loop body's responsibility to free up that storage when it is done with the string.

HLA high-level calling sequence examples:

```

foreach filesys.fileIn( "dirToSearch" ) do

    // At this point, EAX points at a filename of a file
    // found in the "dirToSearch" directory.

    mov( eax, filename );// filename:string

    // Do something with the string...

    // When we're done with the string, free it.

    str.free( filename );

endfor;

```

#### **iterator filesys.fileInCwd;**

This iterator, used within a foreach loop, will repeat once for each file in the current directory. On each iteration of the foreach loop, this function will provide a pointer to the full filename in the EAX register. Note that this function only iterates on regular files, it will not iterate (nor return the string name) of any directory entries. The *filesys.fileInCwd* function will allocate storage for the filename string on the heap and return the pointer to this string in EAX. It is the foreach loop body's responsibility to free up that storage when it is done with the string.

HLA high-level calling sequence examples:

```

foreach filesys.fileInCwd() do

    // At this point, EAX points at a filename of a file
    // found in the current working directory.

    mov( eax, filename );// filename:string

    // Do something with the string...

    // When we're done with the string, free it.

    str.free( filename );

endfor;

```

#### **iterato**

This iterator, used within a foreach loop, will repeat once for each directory entry, whose name ends with *suffix*, in the directory specified by the *directory* parameter. On each iteration of the foreach loop, this function will provide a pointer to the full directory name in the EAX register. Note that this function only iterates on directory files, it will not iterate (nor return the string name) of any regular file entries. The *filesys.dirWithSuffix* function will allocate storage for the directory name string on the heap and return the pointer to this string in EAX. It is the foreach loop body's responsibility to free up that storage when it is done with the string.

HLA high-level calling sequence examples:

```

foreach filesys.dirWithSuffix( "dirToSearch", "suffix" ) do

    // At this point, EAX points at name of a directory
    // that ends with "suffix" that was found in the "dirToSearch"
    // directory.

    mov( eax, dirname );// dirname :string

    // Do something with the string...

```

```

    // When we're done with the string, free it.

    str.free( dirname );

endfor;

```

#### **iterator filesys.dirIn( directory:string);**

This iterator, used within a foreach loop, will repeat once for each directory entry in the directory specified by the *directory* parameter. On each iteration of the foreach loop, this function will provide a pointer to the full directory name in the EAX register. Note that this function only iterates on directory files, it will not iterate (nor return the string name) of any regular file entries. The *filesys.dirIn* function will allocate storage for the directory string on the heap and return the pointer to this string in EAX. It is the foreach loop body's responsibility to free up that storage when it is done with the string.

HLA high-level calling sequence examples:

```

foreach filesys.dirIn( "dirToSearch" ) do

    // At this point, EAX points at name of a directory
    // that was found in the "dirToSearch" directory.

    mov( eax, dirname );// dirname :string

    // Do something with the string...

    // When we're done with the string, free it.

    str.free( dirname );

endfor;

```

#### **iterator filesys.dirInCwd;**

This iterator, used within a foreach loop, will repeat once for each directory entry in the current directory. On each iteration of the foreach loop, this function will provide a pointer to the full directory name in the EAX register. Note that this function only iterates on directory files, it will not iterate (nor return the string name) of any regular file entries. The *filesys.dirInCwd* function will allocate storage for the directory string on the heap and return the pointer to this string in EAX. It is the foreach loop body's responsibility to free up that storage when it is done with the string.

HLA high-level calling sequence examples:

```

foreach filesys.dirInCwd() do

    // At this point, EAX points at name of a directory
    // that was found in the current working directory.

    mov( eax, dirname );// dirname :string

    // Do something with the string...

    // When we're done with the string, free it.

    str.free( dirname );

endfor;

```

```
iterator filesys.itemWithSuffix( directory:string; suffix:string );
```

This iterator, used within a foreach loop, will repeat once for each entry, whose name ends with *suffix*, in the directory specified by the *directory* parameter. On each iteration of the foreach loop, this function will provide a pointer to the name in the EAX register. Note that this function iterates on both directory and regular file entries. The *filesys.itemWithSuffix* function will allocate storage for the result string on the heap and return the pointer to this string in EAX. It is the foreach loop body's responsibility to free up that storage when it is done with the string.

HLA high-level calling sequence examples:

```
foreach filesys.itemWithSuffix( "dirToSearch", "suffix" ) do

    // At this point, EAX points at a filename or a directory
    // name ending with "suffix"

    mov( eax, entryname );// entryname :string

    // Do something with that string...

    // When we're done with the string, free it.

    str.free( entryname );

endfor;
```

```
iterator filesys.itemInCwd;
```

This iterator, used within a foreach loop, will repeat once for each entry in the current directory. On each iteration of the foreach loop, this function will provide a pointer to the name in the EAX register. Note that this function iterates on both directory and regular file entries. The *filesys.itemInCwd* function will allocate storage for the result string on the heap and return the pointer to this string in EAX. It is the foreach loop body's responsibility to free up that storage when it is done with the string.

HLA high-level calling sequence examples:

```
foreach filesys.itemInCwd( ) do

    // At this point, EAX points at a filename or a directory
    // name from the current working directory.

    mov( eax, entryname );// entryname :string

    // Do something with that string...

    // When we're done with the string, free it.

    str.free( entryname );

endfor;
```



## 17 HLA Related Macros and Constants (hla.hhf)

The HLA module contains numeric constants produced by some of the HLA symbol-table compile-time functions. It also contains various macros to extend the HLA compile-time language and provide support for other HLA stdlib modules.

### 17.1 The HLA Module

To use the HLA macros and constants in your application, you will need to include one of the following statements at the beginning of your HLA application:

```
#include( "hla.hhf" )
or
#include( "stdlib.hhf" )
```

### 17.2 Classification Macros

The hla.hhf module contains some macros that test the type of an identifier at compile time. Here is a typical invocation of these macros:

```
#if( hla.IsUns( uVar ) )
    // do something if Uns object
#else
    // Do something if not unsigned object
#endif
```

**#macro hla.IsUns( identifier );**

This macro returns a compile-time expression that evaluates true if the specified identifier is an uns8, uns16, uns32, uns64, or uns128 object.

**#macro hla.IsInt( identifier );**

This macro returns a compile-time expression that evaluates true if the specified identifier is an int8, int16, int32, int64, or int128 object.

**#macro hla.IsHex( identifier );**

This macro returns a compile-time expression that evaluates true if the specified identifier is a byte, word, dword, qword, tbyte, or lword object.

**#macro hla.IsNumber( identifier );**

This macro returns a compile-time expression that evaluates true if the specified identifier is an uns8, uns16, uns32, uns64, uns128, int8, int16, int32, int64, int128, byte, word, dword, qword, tbyte, or lword object.

**#macro hla.IsReal( identifier );**

This macro returns a compile-time expression that evaluates true if the specified identifier is a real32, real64, or real80 object.

```
#macro hla.IsNumeric( identifier );
```

This macro returns a compile-time expression that evaluates true if the specified identifier is an uns8, uns16, uns32, uns64, uns128, int8, int16, int32, int64, int128, byte, word, dword, qword, tbyte, lword, real32, real64, or real80 object.

```
#macro hla.IsOrdinal( identifier );
```

This macro returns a compile-time expression that evaluates true if the specified identifier is an uns8, uns16, uns32, uns64, uns128, int8, int16, int32, int64, int128, boolean, char, byte, word, dword, qword, tbyte, lword, or enumerated data type object.

## 17.3 String to Integer Macros

The HLA module provides two macros, `hla.asWord` and `hla.asDword`, that let you treat a one to four-character string as a 16-bit or 32-bit number. Specifically, these macros cram the 1-4 bytes of the strings into a two-byte word constant or a 4-byte dword constant.

```
#macro hla.asWord( "1 or 2 character string" );
```

This macro places the first character of the string in the L.O. byte of the 16-bit result, and the second character of the string in the H.O. byte of the result. If there is only one character in the string, the H.O. byte of the result will be zero.

```
#macro hla.asDword( "1 to 4 character string" );
```

This macro places copies the first through fourth characters of the string to the L.O. to H.O. bytes of the dword results. If there are fewer than 4 characters, the H.O. byte(s) of the result are filled with zeros.

## 17.4 Label Generation Macro

The `hla.genlabel` macro generates a sequence of strings that are unique, legal, HLA identifiers (within the current compilation, do not use these as public symbols). Typically, you would take the string that this macro returns and convert that string to an actual symbol using the `@TEXT` function.

Here's the definition of `hla.getLabel` in the HLA header file:

```
val
    _hla_labelCnt_ := 0;

#macro genLabel;

    "_genLabel_" + string( hla._hla_labelCnt_ ) + "_"
    ?hla._hla_labelCnt_ := hla._hla_labelCnt_ + 1;

#endmacro;
```

## 17.5 Procedure Overloading Macro

The `hla.overload` macro allows you to provide the equivalent of "overloaded functions" in your HLA programs. Note that this macro is somewhat obsolete as the HLA language now provides built-in overloading (see the HLA Reference Manual for details). This macro is provided for compatibility with old code that is still using it.

Overloaded functions are called by "signature" rather than by name. A signature is the combination of the name of a function, the number of parameters, and the types of each of the parameters. As long as two separate functions have unique signatures, they can be called using the same name. Of course, HLA requires each

procedure/iterator/method/macro to have a unique name within the current scope, but the overload macro lets you define the name to use when calling these different functions.

As an example, consider the following three function prototypes:

```
procedure min8( val1:uns8; val2:uns8 ); @external;
procedure min16( val1:uns16; val2:uns16 ); @external;
procedure min32( val1:uns32; val2:uns32 ); @external;
```

We would like to call these three functions using the single name "min" and letting the different signatures (the differing parameter types in this case) select the actual function to call. The `hla.overload` macro will write a "min" macro for us that will determine the parameter types and call one of the functions above based on the signature of the call. An overloaded definition looks like the following:

```
#macro min( _parms_[] );// Define "min", our overloaded procedure
  hla.overload( _parms_ );
    hla.signature( min8( uns8, uns8 ) );
    hla.signature( min16( uns16, uns16 ) );
    hla.signature( min32( uns32, uns32 ) );
  hla.endoverload
#endmacro
```

The first thing to note is that you must create a macro that will be used as the "overloaded procedure". In this example, that macro is the "min" macro. This macro must always have a single array (variable) parameter. The name isn't important, though "`_parms_`" is the conventional name to use here.

The body of the overloaded macro is actually going to be written by the `hla.overload` macro. The `hla.overload` macro is an HLA context-free macro that begins with "`hla.overload`", ends with "`hla.endoverload`", and contains a single "`hla.signature`" keyword macro invocation for each signature we want our overloaded procedure to support. In this example, we want our min function to call one of three different functions based on the types of the parameters passed to min, so there will be three signature invocations.

A signature keyword macro invocation takes the following form:

```
hla.signature( actualProcedureName( list_of_parameter_types ) );
```

where *actualProcedureName* is the name of the function we want to call if the signature matches and *list\_of\_parameter\_types* is a comma-separated list of HLA data type names, that correspond to the data types for the signature.

The `hla.overload` macro will write the body of the min macro so that it will parse the parameters passed in (via `_parms_`) and determine which of the three functions to call. For example:

```
min( u8, u8a ); // calls min8( u8, u8a );
min( u16, u16a ); // calls min16( u16, u16a );
min( u32, u32a ); // calls min32( u32, u32a );
```

Here are some additional examples of overloaded macro definitions from the HLA stdlib:

```
#macro catsub( parms[] );

  hla.overload( parms )

    hla.signature( str.catsub4(string, dword, dword, string) )
    hla.signature( str.catsub5(string, dword, dword, string, string))

  hla.endoverload

#endmacro

procedure catsub4( src:string; start:dword; len:dword; dest:string );
  external( "STR_CATSUB4" );

procedure catsub5
(
  src2:string;
```

```

    start:dword;
    len      :dword;
    src1:string;
    dest:string
);
@returns( "(type string eax)" );
external( "STR_CATSUB5" );

#macro first( parms[] );

    hla.overload( parms )

        hla.signature( str.first2(string, dword) )
        hla.signature( str.first3(string, dword, string) )

    hla.endoverload

#endmacro

procedure first2
(
    s      :string;
    len    :dword
); external( "STR_FIRST2" );

procedure first3
(
    s      :string;
    len    :dword;
    dest:string
); external( "STR_FIRST3" );

```

When you use an overloaded function (that is, you invoke the macro whose body was filled in by the *hla.overload* macro), the code attempts to match the actual parameters against the signatures you've provided. If no possible signature can match the actual parameter list, then the system will report an error. If two or more signatures can be matched by the actual parameter list, the system will also report an error. Unfortunately, if the actual parameter list provides an ambiguous footprint, then the system will report an error. Consider the following overloaded function and a couple invocations of that function:

```

static
    u8      :uns8;
    u16 :uns16;
    s      :string;

#macro abc( _parms_[] );
    hla.overload( _parms_ )

        hla.signature( abc1( uns8, string ) )
        hla.signature( abc2( uns16, string ) )

    hla.endoverload;
#endmacro

.
.
.
abc( u8, s ); // Okay, calls abc1
abc( u8, "abc1" ); // Okay, calls abc1
abc( u16, s ); // Okay, calls abc2

```

```
abc( 0, s );// Illegal - matches multiple signatures
```

The last example doesn't work because the literal constant zero matches both the `uns8` and `uns16` types, so this is an ambiguous signature match. Keep this limitation in mind when creating your signatures.

## 17.6 Generic PUT Macro

The *hla.put* macro provides a mechanism for creating generic "put" functions like the *stdout.put*, *stderr.put*, *fileio.put*, and *str.put* macros found in the HLA standard library. Indeed, these macros were built using the *hla.put* macro. By using the *hla.put* macro, you can easily create your own "put" macro that calls a variety of different functions based on the types of the parameters passed to your "put" macro.

First of all, it's important to realize that a user-defined "put" macro must appear inside a namespace. The *hla.put* macro expects to find several objects defined in a namespace whose name you provide. The namespace will contain all the procedures that the user-defined macro will ultimately call as well as a few additional compile-time data structures that the *hla.put* macro will reference.

Within the namespace you're defining a "put" macro for, there must be six constant array objects with *hla.sizePTypes* elements each. These arrays must have the following names and types:

```
validPutType:boolean [ @global:hla.sizePTypes ];
validPutSizeType:boolean [ @global:hla.sizePTypes ];
validPutSize2Type:boolean [ @global:hla.sizePTypes ];
putFunc          :string [ @global:hla.sizePTypes ];
putSizeFunc:string [ @global:hla.sizePTypes ];
putSize2Func:string [ @global:hla.sizePTypes ];
```

Assuming you've created a namespace called "myns", then *myns.validPutType* will tell the *hla.put* macro which built-in data types it can process when you specify an unadorned argument to *myns.put*. For example, if *i8* is an `int8` object, then *myns.put( i8 );* will call some function if the *myns.validPutType* entry indexed by *hla.ptInt8* contains true. Here is the *validPutType* table for the *stdout* namespace; most *validPutType* tables will be a copy of this one (assuming you want your new "put" macro to handle all the same data types as *stdout.put*):

```
const
  validPutType :boolean[ @global:hla.sizePTypes ] :=
  [
    @global:false, // Undefined
    @global:true,  // tBoolean //1
    @global:false, // enum//2
    @global:true,  // tUns8, //3
    @global:true,  // tUns16, //4
    @global:true,  // tUns32, //5
    @global:true,  // tUns64, //6
    @global:true,  // tUns128, //7
    @global:true,  // tByte, //8
    @global:true,  // tWord, //9
    @global:true,  // tDWord, //10
    @global:true,  // tQWord, //11
    @global:true,  // tTByte, //12
    @global:true,  // tLWord, //13
    @global:true,  // tInt8, //14
    @global:true,  // tInt16, //15
    @global:true,  // tInt32, //16
    @global:true,  // tInt64, //17
    @global:true,  // tInt128, //18
    @global:true,  // tChar, //19
    @global:false, // tWChar, //20
    @global:true,  // tReal32, //21
    @global:true,  // tReal64, //22
    @global:true,  // tReal80, //23
    @global:false, // tReal128, //24
    @global:true,  // tString, //25
    @global:false, // tZString, //26
```

```

    @global:false, // tWString, //27
    @global:true, // tCset, //28
    @global:false, // tArray, //29
    @global:false, // tRecord, //30
    @global:false, // tUnion, //31
    @global:false, // tRegex, //32
    @global:true, // tClass, //33
    @global:false, // tProcptr, //34
    @global:false, // tThunk, //35
    @global:true, // tPointer, //36
    @global:false, // tLabel, //37
    @global:false, // tProc, //38
    @global:false, // tMethod, //39
    @global:false, // tClassProc, //40
    @global:false, // tClassIter, //41
    @global:false, // tIterator, //42
    @global:false, // tProgram, //43
    @global:false, // tMacro, //44
    @global:false, // tText, //45
    @global:false, // tRegExMac, //46
    @global:false, // tNamespace, //47
    @global:false, // tSegment, //48
    @global:false, // tAnonRec, //49
    @global:false, // tAnonUnion, //50
    @global:false, // tVariant, //51
    @global:false, // tError, //52
];

```

Notice that each element of the array is indexed by the pType value for the data type.

The `validPutSizeType` array is very similar to the `validPutType` array (indeed, it is structurally identical to the `validPutType` array). The difference between the two is that the `hla.put` macro uses the `validPutSizeType` array to determine if it can call a `"*Size"` function when the `"put"` macro encounters an operand of the form `"xx:ss"`, where `"ss"` is a print width specification. For example, if you write `"myns.put( i8:4 );"` and `i8` is an `int8` variable, then the `hla.put` macro will check the `hla.ptInt8` element of the `validPutSizeType` array to determine whether it supports a field width for `int8` output. Here is the `stdout` version of this array (again, most uses of the `hla.put` macro will copy this, assuming they provide all the same output functionality as `stdout.put`):

```

validPutSizeType :boolean[ @global:hla.sizePTypes ] :=
[
    @global:false, // Undefined
    @global:true, // tBoolean //1
    @global:false, // enum //2
    @global:true, // tUns8, //3
    @global:true, // tUns16, //4
    @global:true, // tUns32, //5
    @global:true, // tUns64, //6
    @global:true, // tUns128, //7
    @global:true, // tByte, //8
    @global:true, // tWord, //9
    @global:true, // tDWord, //10
    @global:true, // tQWord, //11
    @global:true, // tTByte, //12
    @global:true, // tLWord, //13
    @global:true, // tInt8, //14
    @global:true, // tInt16, //15
    @global:true, // tInt32, //16
    @global:true, // tInt64, //17
    @global:true, // tInt128, //18
    @global:true, // tChar, //19
    @global:false, // tWChar, //20
    @global:true, // tReal32, //21

```

```

@global:true, // tReal64, //22
@global:true, // tReal80, //23
@global:false, // tReal128, //24
@global:true, // tString, //25
@global:false, // tZString, //26
@global:false, // tWString, //27
@global:false, // tCset, //28
@global:false, // tArray, //29
@global:false, // tRecord, //30
@global:false, // tUnion, //31
@global:false, // tRegex, //32
@global:false, // tClass, //33
@global:false, // tProcptr, //34
@global:false, // tThunk, //35
@global:true, // tPointer, //36
@global:false, // tLabel, //37
@global:false, // tProc, //38
@global:false, // tMethod, //39
@global:false, // tClassProc, //40
@global:false, // tClassIter, //41
@global:false, // tIterator, //42
@global:false, // tProgram, //43
@global:false, // tMacro, //44
@global:false, // tText, //45
@global:false, // tRegexMac, //46
@global:false, // tNamespace, //47
@global:false, // tSegment, //48
@global:false, // tAnonRec, //49
@global:false, // tAnonUnion, //50
@global:false, // tVariant, //51
@global:false, // tError, //52
];

```

The `validPutSize2Type` array is very similar to the `validPutSizeType` array. The difference between the two is that the `hla.put` macro uses the `validPutSize2Type` array to determine if it can call a `"*Size"` function when the `"put"` macro encounters an operand of the form `"xx:ww:dd"`, where `"ww"` is a print width specification and `"dd"` is a "number of decimal positions" value. This functionality is typically used for outputting real values in decimal form. For example, if you write `"myns.put( r80:14:2 );"` and `r80` is an `real80` variable, then the `hla.put` macro will check the `hla.ptReal80` element of the `validPutSize2Type` array to determine whether it supports a field width and decimal count for `real80` output. Here is the `stdout` version of this array (again, most uses of the `hla.put` macro will copy this, assuming they provide all the same output functionality as `stdout.put`):

```

validPutSize2Type :boolean[ @global:hla.sizePTypes ] :=
[
    @global:false, // Undefined
    @global:false, // tBoolean //1
    @global:false, // enum //2
    @global:false, // tUns8, //3
    @global:false, // tUns16, //4
    @global:false, // tUns32, //5
    @global:false, // tUns64, //6
    @global:false, // tUns128, //7
    @global:false, // tByte, //8
    @global:false, // tWord, //9
    @global:false, // tDWord, //10
    @global:false, // tQWord, //11
    @global:false, // tTByte, //12
    @global:false, // tLWord, //13
    @global:false, // tInt8, //14
    @global:false, // tInt16, //15
    @global:false, // tInt32, //16

```

```

    @global:false, // tInt64, //17
    @global:false, // tInt128, //18
    @global:false, // tChar, //19
    @global:false, // tWChar, //20
    @global:true, // tReal32, //21
    @global:true, // tReal64, //22
    @global:true, // tReal80, //23
    @global:false, // tReal128, //24
    @global:false, // tString, //25
    @global:false, // tZString, //26
    @global:false, // tWString, //27
    @global:false, // tCset, //28
    @global:false, // tArray, //29
    @global:false, // tRecord, //30
    @global:false, // tUnion, //31
    @global:false, // tRegEx, //32
    @global:false, // tClass, //33
    @global:false, // tProcptr, //34
    @global:false, // tThunk, //35
    @global:false, // tPointer, //36
    @global:false, // tLabel, //37
    @global:false, // tProc, //38
    @global:false, // tMethod, //39
    @global:false, // tClassProc, //40
    @global:false, // tClassIter, //41
    @global:false, // tIterator, //42
    @global:false, // tProgram, //43
    @global:false, // tMacro, //44
    @global:false, // tText, //45
    @global:false, // tRegExMac, //46
    @global:false, // tNamespace, //47
    @global:false, // tSegment, //48
    @global:false, // tAnonRec, //49
    @global:false, // tAnonUnion, //50
    @global:false, // tVariant, //51
    @global:false, // tError, //52
];

```

The *putFunc*, *putSizeFunc*, and *putSize2Func* arrays in your namespace must contain the names of the functions to call if the corresponding entries in *validPutType*, *validPutSizeType*, and *validPutSize2Type* contain true (respectively). These strings must be the name of the procedure to call without the namespace prefix. That is, if you want to tell hla.put to call "myns.puti8" to print an 8-bit integer, then the entry at index *hla.ptInt8* in *myns.putFunc* should contain the string "puti8". Note that if the corresponding entry in the *validPutType*, *validPutSizeType*, or *validPutSize2Type* tables contain false, then it doesn't matter what string appears in the array as *hla.put* will never use it; by convention, the empty string is always put in unused entries. Here is the *stdout.putFunc* table:

```

putFunc :string[ @global:hla.sizePTypes ] :=
[
    "", // Undefined
    "putbool", // tBoolean //1
    "", // enum //2
    "putu8", // tUns8, //3
    "putu16", // tUns16, //4
    "putu32", // tUns32, //5
    "putu64", // tUns64, //6
    "putu128", // tUns128, //7
    "putb", // tByte, //8
    "putw", // tWord, //9
    "putd", // tDWord, //10

```



```

"putq",    // tQWord, //11
"puttb",   // tLWord, //12
"putl",    // tLWord, //13
"puti8",   // tInt8,  //14
"puti16",  // tInt16, //15
"puti32",  // tInt32, //16
"puti64",  // tInt64, //17
"puti128", // tInt128, //18
"putc",    // tChar,  //19
"",        // tWChar, //20
"_pute32", // tReal32, //21
"_pute64", // tReal64, //22
"_pute80", // tReal80, //23
"",        // tReal128, //24
"puts",    // tString, //25
"putz",    // tZString, //26
"",        // tWString, //27
"putcset", // tCset,  //28
"",        // tArray,  //29
"",        // tRecord, //30
"",        // tUnion,  //31
"",        // tRegEx,  //32
"",        // tClass,  //33
"",        // tProcptr, //34
"",        // tThunk,  //35
"putd",    // tPointer, //36
"",        // tLabel,  //37
"",        // tProc    //38
"",        // tMethod,  //39
"",        // tClassProc, //40
"",        // tClassIter, //41
"",        // tIterator, //42
"",        // tProgram, //43
"",        // tMacro,   //44
"",        // tText    //45
"",        // tRegExMac, //46
"",        // tNamespace, //47
"",        // tSegment, //48
"",        // tAnonRec,  //49
"",        // tAnonUnion, //50
"",        // tVariant,  //51
"",        // tError,   //52
];
ut.putFunc table:

```

Please see the `stdout.hhf` header file for examples of the other two tables.

Once you have set up the six constant arrays in your namespace, defining your own `put` macro using *hla.put* is almost trivial. The *hla.put* macro processes a single "put" operand. The syntax for this macro is the following:

```
hla.put( <namespaceID>, <first param as string>, <single argument> );
```

where `<namespace>` is your namespace identifier, `<first param as string>` is the first parameter to be passed to all your functions (this can be the empty string if you don't have a first parameter [e.g., *stdout* or *stderr*], or it can be whatever your output functions require; for example, the *fileio.put* macro specifies a file handle here, the *str.put* macro specifies a string variable name here).

Because *hla.put* only handles a single output object, you must provide a simple `put` macro within your namespace that calls *hla.put* for each of the actual arguments passed to your *put* macro. Here's the *stdout* version of that macro:

```
val
```

```

        stdoutParm:string;

#macro put( _parameters_[] );

    #for( @global:stdout.stdoutParm in _parameters_ )

        @global:hla.put( stdout, "", @eval(@global:stdout.stdoutParm) )

    #endfor

#endmacro

```

Here's the fileio version of this macro:

```

val
    _v_      :string;
    _curparm_:string;

#macro put( _ileVar_, _parameters_[] );

    ?@global:fileio._v_ := @string:_ileVar_;
    #for( @global:fileio._curparm_ in _parameters_ )

        @global:hla.put
        (
            fileio,
            @global:fileio._v_,
            @eval(@global:fileio._curparm_)
        )

    #endfor

#endmacro

```

## 17.7 @class Constants

The hla.hhf module contains some macros that test the type of an identifier at compile time. Here is a typical invocation of these macros:

The HLA compile-time `@class` function returns the following values to denote the classification of an identifier. If a symbol appears more than once in a program, the `@class` function returns the classification value for the symbol currently in scope.

Ταβλε 1 **@Class Return Values**

Name	Value	Description
hla.cIllegal	0	Symbol doesn't have a legal HLA classification.
hla.cConstant	1	Symbol was defined in the CONST section.
hla.cValue	2	Symbol was defined in the VAL section.
hla.cType	3	Symbol was defined in the TYPE section.
hla.cVar	4	Symbol was defined in the VAR section

hla.cParm	5	Symbol is a parameter.
hla.cStatic	6	Symbol was defined in a STATIC, READONLY, or STORAGE section.
hla.cLabel	7	Symbol is a statement label.
hla.cProc	8	Identifier is the name of a (non-class) procedure.
hla.cIterator	9	Identifier is the name of a (non-class) iterator.
hla.cClassProc	10	Identifier is the name of a class procedure.
hla.cClassIter	11	Identifier is the name of a class iterator.
hla.cMethod	12	Identifier is the name of a class method.
hla.cMacro	13	Symbol is a macro.
hla.cKeyword	14	Symbol is an HLA reserved word.
hla.cTerminator	15	Symbol is an HLA TERMINATOR macro.
hla.cRegex	16	Symbol is a regular expression macro
hla.cProgram	17	PROGRAM or UNIT identifier.
hla.cNamespace	18	Identifier is a name space ID.
hla.cSegment	19	Identifier is a segment name.
hla.cRegister	20	Identifier is an 80x86 register name.
hla.cNone	21	Reserved.

## 17.8 HLA pType Constants

The HLA @ptype compile-time function returns the values in the following table for the symbol you pass as a parameter to the function. You should always use these symbol names rather than hard-coding the constants in your programs. These values have changed in the past and they will likely change in the future with improvements to the HLA language.

Ταβλε 2: @pType Return Values

Symbol	Value	Description
hla.ptIllegal	0	Symbol is undefined or is not an object to which a type can be applied.
hla.ptBoolean	1	Symbol is of type boolean.
hla.ptEnum	2	Symbol is an enumerated type.
hla.ptUns8	3	Symbol is an UNS8 object.
hla.ptUns16	4	Symbol is an UNS16 object.

hla.ptUns32	5	Symbol is an UNS32 object.
hla.ptUns64	6	Symbol is an UNS64 object.
Hla.ptUns128	7	Symbol is an UNS128 object.
hla.ptByte	8	Symbol is a BYTE object.
hla.ptWord	9	Symbol is a WORD object.
hla.ptDWord	10	Symbol is a DWORD object.
hla.ptQWord	11	Symbol is a QWORD object.
hla.ptTByte	12	Symbol is a TBYTE object.
hla.ptLWord	13	Symbol is a LWORD object.
hla.ptInt8	14	Symbol is an INT8 object.
hla.ptInt16	15	Symbol is an INT16 object.
hla.ptInt32	16	Symbol is an INT32 object.
hla.ptInt64	17	Symbol is an INT64 object.
hla.ptInt128	18	Symbol is an INT128 object.
hla.ptChar	19	Symbol is of type CHAR.
hla.ptWChar	20	Symbol is of type WCHAR.
hla.ptReal32	21	Symbol is a REAL32 object.
hla.ptReal64	22	Symbol is a REAL64 object.
hla.ptReal80	23	Symbol is a REAL80 object.
hla.ptReal128	24	Symbol is a REAL128 object.
hla.ptString	25	Symbol has the STRING type.
hla.ptZString	26	Symbol has the ZSTRING type.
hla.ptWString	27	Symbol has the WSTRING type.
hla.ptCset	28	Symbol's type is CSET.
hla.ptArray	29	The symbol is an array object.
hla.ptRecord	30	The symbol is a record object.
hla.ptUnion	31	The symbol is a union object.
hla.ptRegex	32	The symbol is a regular expression object.
hla.ptClass	33	The symbol is a class object.
hla.ptProcptr	34	The symbol's type is "pointer to a procedure".
hla.ptThunk	35	The symbol is a THUNK type.
hla.ptPointer	36	The symbol is a POINTER object.

hla.ptLabel	37	The symbol is a statement label object.
hla.ptProc	38	The symbol denotes a procedure.
hla.ptMethod	39	The symbol denotes a method.
hla.ptClassProc	40	The symbol is a procedure within a class.
hla.ptClassIter	41	The symbol denotes an iterator within a class.
hla.ptIterator	42	The symbol is an iterator name.
hla.ptProgram	43	The symbol is the program's or unit's identifier.
hla.ptMacro	44	The identifier is a macro.
hla.ptText	45	The identifier is a text object (note: @ptype does not return this value since HLA expands the text prior to processing by @ptype).
Hla.ptRegExMac	46	The symbol is a regular expression macro.
hla.ptNamespace	47	The identifier is a namespace ID.
hla.ptSegment	48	The identifier is a segment ID.
hla.ptAnonRec	49	The identifier is an anonymous record within a union (internal use only, @ptype will never return this value).
hla.ptAnonUnion	50	The identifier is an anonymous union within a record (internal use only, @ptype will never return this value).
hla.ptVariant	51	This value is reserved for internal use by HLA.
hla.ptError	52	This value indicates a cascading error in an expression. Generally, you will not get this value from @ptype unless there was some sort of error in the parameter to pass to @ptype.

Note that the HLA module provides a constant array, `hla.ptypeStrs`, that returns the associated HLA type name when indexed by one of the above constants. Note that this is a compile-time constant array, not a run-time array of strings. If you want a run-time string array, you can define one thusly:

```
static
  ptypeStrs:string [elements( hla.ptypeStrs )] := hla.ptypeStrs;
```

Do note that not every ptype value maps to a valid HLA data type. For example, `hla.ptRecord` maps to the string "(record)". You cannot use this as a type in an HLA program.

## 17.9 @pclass Return Values

The HLA `@pClass` function expects a procedure's parameter name as its sole parameter. It returns one of the following constants that denotes the parameter passing mechanism for the parameter. Note that `@pClass`' return values are defined only for parameter identifiers. These values have changed in the past and they will likely change in the future with improvements to the HLA language, so always use these symbolic names rather than hard-coded values.

Ταβλ. 3 @pClass Return Values

Symbol	Value	Description
hla.illegal_pc	0	May be returned if the symbol is not a parameter.
hla.valp_pc	1	Returned if parameter is passed by value.
hla.refp_pc	2	@pClass returns this value if you pass the parameter by reference.
hla.vrp_pc	3	Denotes that you've passed the parameter by value/result.
hla.result_pc	4	This value means that you've passed the parameter by result.
hla.name_pc	5	This value indicates that you've passed the parameter by name.
hla.lazy_pc	6	This value indicates that you've passed the parameter by lazy evaluation.

## 17.10 @section Return Values

The following constants correspond to bits in the value returned by @section. They denote the current position of the compiler in the code. These values have changed in the past and they will likely change in the future with improvements to the HLA language, so always use these symbolic names rather than hard-coded values.

Ταβλ. 4 @section Constants

Symbol	Value	Description
hla.inConst	1	Bit zero is set if HLA is current processing definitions in a CONST section.
hla.inVal	2	Bit one is set if HLA is current processing definitions in a VAL section.
hla.inType	4	Bit two is set if HLA is current processing definitions in a TYPE section.
hla.inVar	8	Bit three is set if HLA is current processing definitions in a VAR section.
hla.inStatic	\$10	Bit four is set if HLA is current processing definitions in a STATIC section.
hla.inReadonly	\$20	Bit five is set if HLA is current processing definitions in a READONLY section.
hla.inStorage	\$40	Bit six is set if HLA is current processing definitions in a STORAGE section.

hla.inMain	\$1000	Bit 12 is set if HLA is current processing statements in the main program.
hla.inProcedure	\$2000	Bit 13 is set if HLA is current processing statements in a procedure.
hla.inMethod	\$4000	Bit 14 is set if HLA is current processing statements in a method.
hla.inIterator	\$8000	Bit 15 is set if HLA is current processing statements in an iterator.
hla.inMacro	\$1_0000	Bit 16 is set if HLA is current processing statements in a macro.
hla.inKeyword	\$2_0000	Bit 17 is set if HLA is current processing statements in a keyword macro.
hla.inTerminator	\$4_0000	Bit 18 is set if HLA is current processing statements in a terminator macro.
hla.inThunk	\$8_0000	Bit 19 is set if HLA is current processing statements in a thunk's body.
hla.inUnit	\$80_0000	Bit 23 is set if HLA is current processing statements in a unit.
hla.inProgram	\$100_0000	Bit 24 is set if HLA is current processing statements in a program (not a unit).
hla.inRecord	\$200_0000	Bit 25 is set if HLA is current processing declarations in a record definition.
hla.inUnion	\$400_0000	Bit 26 is set if HLA is current processing declarations in a union.
hla.inClass	\$800_0000	Bit 27 is set if HLA is current processing declarations in a class.
hla.inNamespace	\$1000_0000	Bit 28 is set if HLA is current processing declarations in a union.





## 18 HOWL: The HLA Object Windows Library

The HLA Object Windows Library (HOWL) is an application framework for Microsoft Windows that greatly simplifies GUI application development for Windows in assembly language. It lets you declare forms and widgets that describe the visual layout of a GUI application and then you need only write small procedures that handle events associated with your GUI elements. This is far less work than writing a standard Win32 application and processing all the messages that Windows sends to such an application.

Once you master the basics of HOWL, you can write the GUI portion of a Win32 application in a fraction of the time it would take using standard Win32 API programming techniques. Your programs will be easier to read and easier to maintain, as well.

As its name suggests, the HLA *Object* Windows Library makes extensive use of HLA's object-oriented programming facilities. If you are unfamiliar with object-oriented programming, or are uncomfortable with HLA's implementation of object-oriented programming, you should take a look at the chapter on "Classes and Objects" in *The Art of Assembly Language* and the chapter on Object-Oriented Programming in the HLA reference manual.

Note: Successful use of HOWL requires HLA v2.8 or later. If you have an earlier version, you will need to upgrade to the latest version in order to use HOWL. It should be obvious by now, but HOWL for Windows only supports Microsoft Windows (2000, XP, Vista, and Windows 7; no guarantees on earlier versions).

### 18.1 The HOWL Application Framework

Most HLA programs (at least to date) are written with a "main program" that controls the activities of that application by calling various procedures and other code within the application; the main program is the "traffic cop" that determines what happens and the sequence of those operations. An application framework (like HOWL), on the other hand, contains its own "main program" that controls the sequence of operations within the program. As an application programmer, you will supply various procedures that the framework will call and your procedures will handle various tasks as requested by the application framework's main program. This paradigm takes a small amount of effort to get used to if you've never written event-driven applications before, but it's fairly easy to master and you should be able to master it in a few short hours.

For technical reasons, your HOWL applications will still be written as an HLA program. However, the main program for your HOWL application is trivial; if your program is named "howlDemo", then this is what your main program will look like:

```
begin howlDemo;

    HowlMainApp();

end howlDemo;
```

HowlMainApp is the HOWL application framework main program that you must call from your main program to get the application running.<sup>1</sup>

Beyond the main program in your HOWL application, there are two methods and three procedures you must also supply:

- An "onClose" method (usually containing a single statement)
- An "onCreate" method (usually empty)
- An "AppStart" procedure
- An "AppTerminate" procedure
- An "AppException" procedure

Your application will normally contain many more procedures and other code, of course, but every HOWL application will have at least these five procedures and methods.

The first step in creating a HOWL application is to create a subclass of the `wForm_t` type. You could create such a new class using a declaration like the following:

---

1. In theory, the HOWL framework could require that you create a unit and the HOWL main program could have been linked in automatically. However, you cannot insert any main programs into the HLA Standard Library (where HOWL resides) without creating linkage problems for all other applications. The simple alternative is to require HOWL apps to call the `HowlMainApp` procedure from their main program.

```
myNewForm_t:
  class inherits( wForm_t );
    << any new fields you want to add >>
  endclass;
```

However, in a typical HOWL application you won't create a new `wForm_t` type this way. Instead, you'll use the `wForm..endwForm` declaration to achieve this:

```
wForm( myForm )
  << any new fields you want to add   >>
  << any widget declarations you want >>
endwForm
```

The `wForm..endwForm` statement does several things for you:

- It defines a new class type that is a subclass of `wForm_t`. This new class type is given the name "myForm\_t" (substituting whatever name you supply as the argument to `wForm` for "myForm" in this example).
- It creates a static global variable named "`pformname`" (substituting the name you supply in the `wForm` invocation for `formname`) that is a pointer to an object of type `formname_t` (again, substituting the name you supply in the invocation to `wForm` for `formname`).
- It creates a static global variable named "`formname`" (usual substitution) that is an instance of the class object `formname_t`.
- It initializes "`pformname`" with the address of the global "`formname`" variable.
- It creates a macro, named `formname` implementation, that you can invoke to create a constructor for this new class that initializes all the widgets you declare in the `wForm..endwForm` statement.

You could manually do all of these things that the `wForm..endwForm` declaration does for you, but it's a lot easier and less error prone to use the `wForm..endwForm` statement. So this documentation will only consider that approach. If you're interested in learning how to manually create new `wForm_t` subclass types, take a look at the macro implementation of `wForm..endwForm` in the `howl.hhf` header file.

In order to provide concrete examples, this documentation will assume that you're supplying the name "myForm" as an argument to the `wForm..endwForm` macro invocation. Please keep in mind that you can use any name you choose (and "myForm" is not a particularly descriptive or good name) in these examples. In general, you might want to consider using your application's name (if you're not already using that as an identifier elsewhere in your program) as the main form name.

With the basic description of `wForm` out of the way, we can now take a look at a complete HOWL application (the generic equivalent of a "hello world" application in the GUI world). The following code appears in pieces in order to explain the purposes of each piece.

```
program howlDemo;
#linker( "comdlg32.lib" )
#linker( "comctl32.lib" )
```

The HOWL library makes use of the Windows common controls and common dialogs dynamically linked libraries. The two `#linker` statements above instruct HLA to emit instructions to the linker to link in these libraries (so you don't have to specify these library names on the HLA compiler command line).

```
?@NoDisplay      := true;
?@NoStackAlign   := true;
```

By default, HLA emits code to generate displays for all procedures and it also emits code to align the stack at the beginning of each procedure. These options are rarely needed in HOWL programming, so it's a good idea to include the two statements above in order to turn off the code generation for these two features.

```
#includeOnce( "stdlib.hhf" )
#includeOnce( "howl.hhf" )
```

Like most HLA applications, we'll include the HLA standard library generic header file (`stdlib.hhf`) so we can take advantage of most of the standard library's features. Because `stdlib.hhf` doesn't automatically include the `howl.hhf` header file, the code needs to explicitly include this header file as well in order to use HOWL features.

```

static
    align( 4 );
    bkgBrush_g    :dword;
    bkgColor_g    :dword;

```

The `bkgBrush_g` and `bkgColor_g` variables will hold the background color and brush for our forms. We need to declare these variables before our form declaration because many widgets (controls) will need a background color and the widgets will look better if their background color is the same as the form's.

```

wForm( myForm )
endwForm

```

Here's our `wForm` declaration. It is important for you to understand that the `wForm..endwForm` declaration is equivalent to a class declaration in the `type` section of an HLA program. The `wForm` macro, in fact, expands in-place) to a type section with a class declaration for `myForm_t`. There are two things you can place between the `wForm` and `endwForm` clauses: class field declarations and invocations of certain context-free macros defined by the `wForm` macro invocation. We'll take a look at both of these options a little later, for the current example our `wForm..endwForm` macro invocation is empty because we're just creating an empty form for our application. Note, and this is very important, that the `wForm..endwForm` statement must appear at some point in your program where it is legal to begin a type section that defines a class (because this is exactly what `wForm..endwForm` is going to do).

```

// Implement the mainAppWindow create procedure and object instances:

myForm_implementation();

```

One of the tasks that the `wForm..endwForm` macro invocation accomplishes is the creation of a macro (named `myForm_implementation` in this example) that will expand to some code that the `wForm..endwForm` macro generates. Somewhere in your program you must invoke this macro (`myForm_implementation`) in order to actually emit the code that the new class you've created requires. If you're wondering why the `wForm..endwForm` macro doesn't emit this code directly, just keep in mind that the `wForm..endwForm` declaration often appears in a header file (rather than directly in your main program as it appears here) and if `wForm..endwForm` automatically emitted this code, it would be emitted in every file that included the header file. This would result in duplicate code (and duplicate external labels). Therefore, the `wForm..endwForm` statement generates this macro that you must invoke exactly once in your program in order to compile the code that `wForm..endwForm` generates.

```

method myForm_t.onCreate;
begin onCreate;
end onCreate;

method myForm_t.onClose;
begin onClose;

    w.PostQuitMessage( 0 );

end onClose;

```

The `myForm_t` class that `wForm..endwForm` creates will define two methods but it will not generate the code for these methods, you will have to provide the code for these methods. Most of the time, these methods will appear exactly as the two above. The HOWL main program invokes the `onCreate` method at the very end of the execution of the constructor procedure for the `myForm_t` class. You could put code in this method to execute immediately after the creation of the new form but, as you'll soon see, you'll most often put this "on creation" code in the `appStart` procedure. So most of the time the `onCreate` method will be empty. Why does HOWL generate a call to a method you'll almost always leave empty? Well, this is an artifact of object-oriented programming. The `onCreate` method is quite useful for other classes that are derived from `myForm_t`'s parent class, so rather than make a special case out of `myForm_t` (or the other classes), HOWL requires you to write this (usually empty) method and it will call it. Fortunately, this only takes a few bytes of memory and it only gets called once, so there really isn't an efficiency loss for doing this.

When the user of your application clicks on the form's close button, or otherwise tells the application to terminate, HOWL will direct control to the `myForm_t.onClose` method. You can do other things to clean up your application when the program is about to quit, but the main thing you need to do is to tell Windows that the application is quitting and this is done by executing `w.PostQuitMessage( 0 );` As it turns out, you generally won't put any application cleanup code directly in the `onClose` method; there is an `appTerminate` procedure that HOWL will call when your program terminates execution and you'll put all your cleanup code in that procedure.

We've talked about the `appStart` and `appTerminate` procedures, let's take a look at them. A typical `appStart` procedure takes the following form:

```
procedure appStart;
begin appStart;

    push( esi );

    // Create the main application window:

    w.GetSysColor( w.COLOR_MENU );
    mov( eax, bkgColor_g );
    w.CreateSolidBrush( eax );
    mov( eax, bkgBrush_g );
```

The above statements initialize the `bkgColor_g` and `bkgBrush_g` global variables with the color (and corresponding solid brush) of the menu area on a typical Windows' window. Though it's not strictly necessary to have these global variables, they are useful when creating widgets in more complex applications, so most `appStart` procedures will initialize these variables. Note that you must initialize these variables before calling the constructor for the `myForm_t` class because the widget declarations appearing in the `wForm..endwForm` statement will be initialized in the call to the constructor for `myForm`. Note that you don't have to specify the `w.COLOR_MENU` background color. You can specify any RGB color you like. You can use the RGB macro (e.g., `RGB( 255, 128, 0 )`) to specify the individual red, green, and blue (respectively) components of the RGB color.

The `wForm( mForm ) .. endwForm` statement appearing earlier in the source file (and the `myForm_implementation` macro invocation) automatically generated a constructor for the `myForm_t` class named `create_myForm`. That declaration also created an instance variable (an object) of type `myForm_t` named `myForm`. In order to initialize and display the form, you must call this constructor with the following (or reasonably compatible) arguments:

```
myForm.create_myForm
(
    "My Form",           // Window title
    0,                   // Extended style
    0,                   // Style
    NULL,                // No parent window
    w.CW_USEDEFAULT,     // x-position on screen
    w.CW_USEDEFAULT,     // y-position on screen
    600,                 // Width
    600,                 // Height
    bkgColor_g,          // Background color
    true                 // Make visible on creation
);
mov( esi, pmyForm );    // Save pointer to main window object.
```

The first argument is a string parameter that HOWL will display in the title bar of the application's main form. You can put any string you like here, although you should generally put the program's name in the title bar.

The next two parameters let you extend the window style and extended style. In general, this two parameters will contain zero. HOWL always uses the style (`w.WS_CLIPCHILDREN` | `w.WS_OVERLAPPEDWINDOW`) and the extended style `w.WS_EX_CONTROLPARENT` when creating windows for a HOWL form. The styles you specify in the second and third constructor arguments will be logically ORed into HOWL's existing styles.

The fourth parameter is the handle of the form's parent form. For main application windows, this argument is always NULL.

The fifth and sixth arguments in the constructor invocation are the x- and y-coordinates on the screen where Windows will position the upper-left-hand corner of the form. You may specify any reasonable values here (within the range of the size of your screen, if you want the form to be visible) or you can specify `w.CW_USEDEFAULT`, in which case Windows will position the window automatically for you. For the position on the screen, the default coordinates are probably good values to use.

The seventh and eighth arguments are the width and height of the window you're creating. You'll have to pick these numbers based on the layout of the widgets on your form. You can also specify `w.CW_USEDEFAULT` for the width and height and Windows will pick values it feels are appropriate. Generally, though, you'll want to explicitly specify the width and height for a typical form.

The ninth parameter is the background color to use for the main client area of the form. This is an RGB color value. As this example demonstrates, it's a good idea to use the `bkgColor_g` value in this parameter argument so you can specify that same value (or `bkgBrush_g`) for other elements on the display that have a background color.

```
// Return main window handle in EAX if you want the system
// to support control selection via the TAB key. Return
// NULL in EAX if you don't want to support TAB key selection
// of the controls:

mov( myForm.handle, eax );
pop( esi );

end appStart;
```

The caller to `appStart` checks the return value in EAX to determine whether the form supports tab control selection. If you return NULL in EAX, then the form ignores the tab key during execution. If you return the form's handle (`myForm.handle`), then Windows will support control/widget tab selection on the form.

```
// appTerminate-
//
// Called when the application is quitting, giving the app a chance
// to clean up after itself.
//
// Note that this is called *after* the mainAppWindow_t.onClose method
// executes (indeed, mainAppWindow_t.onClose, by posting the quit message,
// is what actually causes the program to begin terminating, which leads
// to the execution of this procedure).

procedure appTerminate;
begin appTerminate;

    // Clean up the main application's form.
    // Note that this will recursively clean up all the widgets on the form.

    myForm.destroy();
    w.DeleteObject( bkgBrush_g );

end appTerminate;
```

The HOWL system will call the `appTerminate` procedure after the `myForm.onClose` method terminates. The `appTerminate` procedure is a good place to call the class destructor (`myForm.destroy`) and clean up any other resources in use. In the code above, for example, the `appTerminate` procedure deletes the brush resource created in the `appStart` procedure. Note that you should only call the main form destructor in the `appTerminate` procedure, not in the `myForm.onClose` method. The destructor will free up any allocated storage and other resources in use and you shouldn't do this while the object is still active (e.g., while the `onClose` method is executing).

```
// appException-
```

```
//
// Gives the application the opportunity to clean up before
// aborting when an unhandled exception comes along:

procedure appException( theException:dword in eax );
begin appException;

    raise( eax );

end appException;
```

HOWL will call the `appException` procedure if an unhandled exception occurs during the execution of your program. As a general rule, it's a good idea to clean up resources as best you can before bailing out of the program.

```
// The main program for a HOWL application must simply
// call the HowlMainApp procedure.

begin howlDemo;

    HowlMainApp();

end howlDemo;
```

The main application of a HOWL program, as noted earlier, should simply call the `HowlMainApp` procedure which is the real “main program” of the system.

Here's the complete “trivial HOWL application” without the annotations:

```
program howlDemo;
#linker( "comdlg32.lib" )
#linker( "comctl32.lib" )

?@NoDisplay      := true;
?@NoStackAlign   := true;

#includeOnce( "stdlib.hhf" )
#includeOnce( "howl.hhf" )

static
    align( 4 );
    bkgBrush_g    :dword;
    bkgColor_g    :dword;

wForm( myForm )
endwForm

// Implement the mainAppWindow create procedure and object instances:

myForm_implementation();

method myForm_t.onClose;
begin onClose;

    w.PostQuitMessage( 0 );
```

```

end onClose;

method myForm_t.onCreate;
begin onCreate;
end onCreate;

procedure appStart;
begin appStart;

    push( esi );

    // Create the main application window:

    w.GetSysColor( w.COLOR_MENU );
    mov( eax, bkgColor_g );
    w.CreateSolidBrush( eax );
    mov( eax, bkgBrush_g );
    myForm.create_myForm
    (
        "My Form",           // Window title
        0,                   // Extended style
        0,                   // Style
        NULL,                // No parent window
        w.CW_USEDEFAULT,     // Let Windows position this guy
        w.CW_USEDEFAULT,     // " " " " " "
        600,                 // Width
        600,                 // Height
        bkgColor_g,          // Background color
        true                 // Make visible on creation
    );
    mov( esi, pmyForm );    // Save pointer to main window object.

    // Return main window handle in EAX if you want the system
    // to support control selection via the TAB key. Return
    // NULL in EAX if you don't want to support TAB key selection
    // of the controls:

    mov( myForm.handle, eax );
    pop( esi );

end appStart;

// appTerminate-
//
// Called when the application is quitting, giving the app a chance
// to clean up after itself.
//
// Note that this is called *after* the mainAppWindow_t.onClose method
// executes (indeed, mainAppWindow_t.onClose, by posting the quit message,
// is what actually causes the program to begin terminating, which leads
// to the execution of this procedure).

procedure appTerminate;
begin appTerminate;

```

```

    // Clean up the main application's form.
    // Note that this will recursively clean up all the widgets on the form.

    myForm.destroy();
    w.DeleteObject( bkgBrush_g );

end appTerminate;

// appException-
//
// Gives the application the opportunity to clean up before
// aborting when an unhandled exception comes along:

procedure appException( theException:dword in eax );
begin appException;

    raise( eax );

end appException;

// The main program for a HOWL application must simply
// call the HowlMainApp procedure.

begin howlDemo;

    HowlMainApp();

end howlDemo;

```

Assuming you've named this file "myForm.hla", you can compile and run this program as follows:

```
hla myForm
```

You can run the application from a command-line by typing "myForm" or you can double-click on the icon to run it in a traditional GUI style. Note that if you double-click on the icon, Windows will open up both the GUI window (your form) and a console window. As it turns out, having the console window is useful for debugging purposes, but if you want to ship a release version of your application, you'll probably want to ditch the console window. You can achieve this by using the following command line to compile your application:

```
hla -w myForm
```

When you run the program (either from the command line or by double clicking on an icon), an empty gray window will appear. You can quit the program by clicking on the close box on the title bar.

To understand how the HOWL application framework functions, we need an example that is slightly more complex than the trivial example presented thus far. Consider the following modifications to the previous example:

```

procedure onPress(thisPtr:dword; wParam:dword; lParam:dword ); forward;

wForm( myForm )

    wPushButton
    (
        myButton,          // Field name in mainWindow object
        "Close",           // Caption for push button
        10,                // x position
        10,                // y position
    )

```



```

        125,                // width
        25,                // height
        onPressed          // "on click" event handler
    );

endwForm
.
.
.
procedure onPressed(thisPtr:dword; wParam:dword; lParam:dword );
begin onPressed;

    w.PostQuitMessage( 0 );

end onPressed;

```

The `onPress` procedure is an example of an *event handler*. The HOWL code will automatically call various event handlers in response to some system event, such as someone pressing a button on a form. In this example, we're going to place a push button on the form and the HOWL code will call the `onPress` event handler in response to someone pressing that button. How does HOWL know to call `onPress` rather than some other function? Easy, we're going to tell it about `onPress`. The first step is to create a forward declaration for `onPress` that must appear before the `wForm..endwForm` statement. We're going to reference `onPress` inside the `wForm..endwForm` statement, so we have to make sure it's declared before `wForm( myForm );` the forward declaration achieves this.<sup>2</sup>

The `wPushButton` declaration appearing inside the `wForm..endwForm` statement tells HOWL to create a push button object on the form. The first argument (`myButton`) is the name of the field that HOWL will create inside the `myForm` class to represent the push button object. This is roughly equivalent to the following declaration in HLA:

```

type
    myForm: class inherits( wForm_t );
    var
        myButton: wPushButton_p;
        .
        .
        .
endclass;

```

(note that type `wPushButton_p` is defined to be a pointer to `wPushButton_t`.)

The remaining arguments in the `wPushButton` statement define the appearance of the button on the actual form. The second argument is the string label that HOWL will display on the button. The third through sixth arguments, as the comments state, specify the x- and y-coordinates and the width and height of the button on the form. Note that the coordinates are relative to the upper-left-hand corner of the *client area* on the form.<sup>3</sup> The last argument is the one of most interest to us here. This is the name of the "on click" event handler that HOWL will call when you press the push button on the form.

All widget event handlers are "widget procs". This means that they are procedures with the following parameter list:

```

type
    widgetProc : procedure( thisPtr:dword; wParam:dword; lParam:dword );

```

Note that `widgetProc` procedures are not class procedures or methods, they are standard procedures. Specifically, this means that you can't use the `this` or `super` reserved words inside a `widgetProc`. However, note that a `widgetProc` does have a `thisPtr` parameter. This parameter will contain the address of the object that signaled the event (that is, `thisPtr` will point at the `myButton` object in this example when the user presses the button and HOWL calls the `onPress` procedure). In this example, we'll not use `thisPtr`, but in more

2. You could also put the entire `onPress` procedure before the `wForm` declaration, but generally it's better programming style to put all procedure code after the declarations (like `wForm`), hence the use of the forward declaration in this code.

3. The client area does not include the title bar nor the menus. If the form is tabbed, it doesn't include the area holding the tabs, either.

complex examples having a pointer to the object turns out to be important. The `wParam` and `lParam` arguments are passed from Windows message procedure on to the `widgetProc` unchanged. For certain widget types, these arguments contain important data. In the current example, we ignore these arguments as well.

For the `onPress` procedure in this example, we simply want to quit the program when the user presses the “close” button. Therefore, this `onPress` procedure executes the same code as the main form’s `onClose` method, that is: `w.PostQuitMessage( 0 );` If you compile this code and run it, you can quit the program by pressing on the “close” button.

In most cases, these event handler procedures are how HOWL communicates with your code. Whenever the user initiates some event by interacting with the widgets on the form, HOWL calls the corresponding widget procedure(s) to let your program handle the event.

## 18.2 The HOWL Declarative Language

In the previous section you saw a couple of simple examples of the HOWL declarative language. The HOWL declarative language is the set of statements that are legal inside a `wForm..endwForm` sequence. The `wPushButton` statement was a good example of a HOWL declarative language statement. There are many other widget-defining statements present in the HOWL declarative language. This section will describe those statements.

Before describing the actual statements that define widgets, you should note that the `wForm..endwForm` statement is actually a class declaration in the HLA language. So in addition to all the legal declarative statements, you can also put any legal class declarations inside a `wForm..endwForm` statement. For example, if you want to communicate some data between various widgets on a form, one way to achieve this is by placing some class `var` declarations inside the `wForm..endwForm` statement:

```
wForm( myForm )
    var
        someData :dword;
        .
        .
        .
endwForm
```

Your event handlers and other code can refer to this data field using the syntax `myForm.someData`.

Of course, you can add any other legal class objects into this declaration including class constants, procedures, methods, and iterators.

The `howl.hhf` header file implements the HOWL declarative language using an HLA context-free macro. The `wForm..endwForm` macro declaration includes various `#keyword` macros (like `wPushButton`) that expand into appropriate declarations in the class (and via some macro magic, store away code to create the constructor procedure for the class). When you look at some HOWL declarative code, it's easy to forget that you're looking at a *declaration*, not at *executable code*. It's easy to think that you should be able to do something like the following:

```
wForm( myForm )

    mov( btnXPosn, eax );
    add( 10, eax );
    mov( btnYPosn, ecx );
    add( 10, ecx );
    wPushButton
    (
        myButton,           // Field name in mainWindow object
        "Close",            // Caption for push button
        eax,                // x position
        ecx,                // y position
        125,                // width
        25,                 // height
        onPress              // "on click" event handler
    );

endwForm
```

However, this won't work. Don't forget that the statements inside the `wForm..endwForm` declaration are emitted inside a class declaration. Stray statements such as `mov( btnXPosn, eax );` cannot appear inside a class declaration.

It is possible to sneak certain statements into your widget declarations. Except for the first parameter in most widget declarations (which is the widget object's name in the class declaration), HOWL records the parameter's value and "plays it back" when generating the code for the form's class constructor. This means you can sneak in some code if that code is appropriate for a parameter in a procedure call. Consider the following:

```
wForm( myForm )

    wPushButton
```

```

    (
        myButton,          // Field name in mainWindow object
        "Close",           // Caption for push button
        returns
        ( {
            mov( btnXPosn, eax );
            add( 10, eax );
        }, "eax" ),
        10,                 // y position
        125,                // width
        25,                 // height
        onPress             // "on click" event handler
    );

endwForm

```

You may specify global variables as arguments to a widget declaration, but you have to ensure that you've declared the global object prior to the `wForm` statement at that you've initialized the global object before calling the constructor for that form.

The following subsections describe the `wForm..endwForm` statement as well as all the widgets that may appear within a `wForm..endwForm` declaration.

## 18.2.1 wForm

```

wForm( <<formname>> )
    << widget and class field declarations >>
endwForm

```

The `wForm..endwForm` statement declares a form (or window) for an application. Every application will have at least one of these statements. The single argument is the name associated with the form. HOWL takes this name and generates five distinct program entities from it:

- A class type named *formname\_t* that represents the form's type and holds the declarations for all the widgets on the form.
- A data type named *formname\_p*,
- A global variable named *formname*, of type *formname\_t*, that is an instance of the form object.
- A global variable named *pformname* of type *formname\_p* that is initialized with the address of the *formname* object.
- A constructor (class procedure) of the name *formname\_t.create\_formname*. You will call this constructor to initialize the *formname* variable object.

Most programs will use the *formname* variable directly and ignore the *pformname* variable. However, *pformname* is available if having a pointer to the object is more convenient than the object itself.

The constructor for the object has the following prototype:

```

procedure formname_t.create_formname
(
    caption      :string;
    exStyle      :dword;
    style        :dword;
    parent       :dword;
    x            :dword;
    y            :dword;
    width        :dword;
    height       :dword;
    fillColor    :dword;
    visible      :boolean
);

```

**caption**    HOWL displays this string in the title bar of the form's window.

<code>exStyle</code>	HOWL logically ORs this MS Windows "extended window style" ( <code>w.WS_EX_*</code> constants) with <code>w.WS_EX_CONTROLPARENT</code> when creating the window for the form.
<code>style</code>	HOWL logically ORs this MS Windows "window style" ( <code>w.WS_*</code> constants) with <code>w.WS_CLIPCHILDREN</code>   <code>w.WS_OVERLAPPEDWINDOW</code> when creating the window for the form.
<code>parent</code>	This is the handle of the parent window for this form. As most <code>wForm</code> objects do not have parent windows, this parameter should contain <code>NULL</code> .
<code>x</code>	The x-coordinate on the screen of the upper-left-hand corner of the form.
<code>y</code>	The y-coordinate on the screen of the upper-left-hand corner of the form.
<code>width</code>	The width of the form (in pixels)
<code>height</code>	The height of the form (in pixels)
<code>fillColor</code>	This is an RGB value that specifies the background color for the form. The red component is the L.O. eight bits of this value, the green component is in bits 8..15 of this value, and the blue component is in bits 16..23. Bits 24..31 should contain zero. The <code>howl.hhf</code> header file defines a macro, <code>RGB</code> , that lets you assemble an RGB value from three constants: <code>RGB( redConst, greenConst, blueConst )</code> . For example, <code>RGB( 255, 0, 0 )</code> corresponds to red with a full intensity.
<code>visible</code>	This is a boolean constant that specifies whether the form will be created in a visible ( <code>true</code> ) or hidden ( <code>false</code> ) state. You can always call the <code>formname.show()</code> method to make a form visible or the <code>formname.hide()</code> method to hide the form at run-time. For the main application window, this field normally contains <code>true</code> .

Generally, *formname* objects are singletons. That is, you typically create only a single instance of a *formname* object. However, as *formname\_t* is a standard HLA data type, there is nothing stopping you from creating multiple objects of this type. Most of the time, however, it would be somewhat confusing to have two instances of the same form on the display at one time. In some instances this is reasonable. For example, if you have a text editor form and you want to allow the user to edit multiple files (or multiple views of the same file) concurrently, it might make sense to have two instances of the same window on the screen at one time.

`wForm..endwForm` declarations are not recursive. That is, you cannot embed a `wForm` inside another `wForm` declaration. `wForm` windows are main application forms and support menus, tabs, and other facilities only possible on the main application form, hence the restriction.

`wForm` objects are examples of *containers* in HOWL. A container, as its name suggests, may contain other objects (specifically widget objects). There are a couple of different kinds of containers in HOWL, but `wForm` containers are special because they can contain two things that other containers cannot: menus and tabs. Therefore, this document will describe these widgets next

## 18.2.2 wMainMenu..endwMainMenu

```
wMainMenu
```

```
<< main menu widget declarations >>
```

```
endwMainMenu
```

A `wForm` object can contain an optional *main menu* widget. This consists of the `wMainMenu..endwMainMenu` statement. If a form contains a main menu widget, the main menu widget must be the first widget declaration appearing on the form. E.g.,

```
wForm( myForm )
```

```
    wMainMenu
```

```
        << menu item declarations >>
```

```
    endwMainMenu
```

```
        << other widget declarations >>
```

```
endwForm
```

Within the `wMainMenu..endwMainMenu` declaration you define the items that appear on the menu. These include menu items, submenus, and separators. The following subsections describe each of these items.

A `wMainMenu` object can be thought of as a container object, albeit a very specialized one. `wMainMenu` objects can contain `wMenuItems`, `wMenuSeparators`, and `wSubMenu` objects. Unlike normal containers, however, `wMainMenu` objects cannot contain arbitrary HOWL objects.

### 18.2.2.1 wMenuItem

A `wMenuItem` declaration defines a single menu item in a main menu or a submenu. This declaration is only legal within a `wMainMenu..endwMainMenu` or `wSubMenu..endwSubMenu` declaration.

```
wMenuItem
(
    menuItemName,
    menuItemChecked,
    menuString,
    menuHandler
);
```

The `menuItemName` argument is an identifier that HOWL inserts into the `wForm` object to represent this particular menu item. You generally won't directly refer to this identifier in your programs, but HOWL requires a field name so you must supply a unique (to the class) identifier here.

The `menuItemCheck` argument is a boolean constant that specifies whether the menu item can contain a check mark next to it in the menu display. If this is true, then the menu item will have the ability to display (or not display) a checkmark next to the menu item. If this field is false, then the menu item will not have this ability. See the discussion of the `wMenuItem_t.checked` method to see how to set or clear this checkmark.

The `menuString` argument is a string constant that specifies the text that HOWL will display for the menu item.

The `menuHandler` argument is either `NULL` or the name of a `widgetProc` that HOWL will call when the user selects this particular menu item. If the argument is `NULL`, HOWL will ignore the selection of the menu item.

HOWL displays menu items across the menu bar on a `wForm` (the menu bar appears immediately below the title bar in the window). Generally, most main menu items are actually submenus, though straight menu items are also legal in and main menu.

### 18.2.2.2 wMenuSeparator

```
wMenuSeparator
```

The `wMenuSeparator` declaration should only appear in a submenu (that is, within a `wSubMenu..endwSubMenu` declaration). This draws a horizontal bar across the menu to separate sets of menu items in a submenu.

### 18.2.2.3 wSubMenu..endwSubMenu

```
wSubenu

    << menu item declarations >>

endwSubmenu
```

Submenu item declarations are syntactically similar to main menu declarations. However, submenus must always appear inside another menu declaration (either a `wMainMenu..endwMainMenu` or some other `wSubMenu..endwSubmenu` declaration). Unlike `wMenuItem` declarations, there is no handler associated with a submenu. HOWL (and Windows) automatically handles all the processing associated with a submenu.

### 18.2.2.4 Menu Example

Here's a complete menu declaration, including submenus within submenus and menu items in the main menu:

```

wForm( mainAppWindow );

wMainMenu;

    wSubMenu( menu_1, "menu1" );

        wMenuItem( menu_1_1, false, "Item_1_1", handler_1_1 );
        wMenuItem( menu_1_2, true, "Item_1_2", handler_1_2 );
        wMenuItem( menu_1_3, true, "Item_1_3", handler_1_3 );
        wMenuSeparator;
        wMenuItem( menu_exit, false, "Exit", exitHandler );

    endwSubMenu;

wSubMenu( menu_2, "menu2" );

    wMenuItem( menu_2_1, false, "Item_2_1", handler_2_1 );
    wSubMenu( menu_2_2, "Menu_2_2" );

        wMenuItem( menu_2_2_1, false, "Item_2_2_1", handler_2_2_1 );
        wMenuItem( menu_2_2_2, false, "Item_2_2_2", handler_2_2_2 );

    endwSubMenu;

endwSubMenu;

wMenuItem( menu_3, false, "menu3", handler_3 );

endwMainMenu;

    << other widget declarations >>

endwForm

```

## 18.2.3 wTab

```

wTab
(
    tabName,      // identifier
    tabString,    // string
    tabHandler,   // widgetProc name or NULL
    bkgColor      // RGB color
)

```

By default, a `wForm` object provides a single surface to which you can attach widgets. The `wTab` declaration allows you to specify multiple surfaces on a form, each user-selectable by clicking on a tab at the top of the form. Like `wForm` objects, `wTab` objects are containers and may contain all the same widgets (except `wMainMenu` items). If you utilize tabs on a `wForm` object, the first `wTab` declaration must appear after the `wMainMenu` declaration (if any) and before any other widgets, e.g.,

```

wForm( mainAppWindow );

wMainMenu;

    wMenuItem( exitMenu, false, "exit", exitHandler );

endwMainMenu;

wTab( tab1, "tab1", NULL, bkgColor_g );

    << widgets associated with tab1 >>

```

The declarations for widgets associated with a tab appear immediately after that tab up to the next tab declaration or the `endwForm` clause. Most forms, if they use tabs, will have at least two tabs. Here's an example declaration of a form with two tabs:

```
wForm( mainAppWindow );

wMainMenu;

    wMenuItem( exitMenu, false, "exit", exitHandler );

endwMainMenu;

wTab( tab1, "tab1", NULL, bkgColor_g );

    wPushButton
    (
        buttonOnTab1,    // Identifier for button
        "Tab1 Button",   // Caption for push button
        10,              // x position
        10,              // y position
        125,             // width
        25,              // height
        onClick1         // "on click" event handler
    );

wTab( tab2, "tab2", NULL, bkgColor_g );

    wPushButton
    (
        buttonOnTab2,    // Identifier for button
        "Tab2 Button",   // Caption for push button
        10,              // x position
        10,              // y position
        125,             // width
        25,              // height
        onClick2         // "on click" event handler
    );

endwForm
```

This example creates two tab pages on the form, each with on button on the respective forms.

HOWL and Windows automatically handle switching from one form to the other when the user clicks on the tabs.

Note that if you place on or more tabs on a form, the size of the client area (where you can put other widgets) is reduced by the size of the tabs bar at the top of the form.

## 18.2.4 Check Boxes

HOWL supports five types of check boxes: generic check boxes (`wCheckable`), standard check boxes (`wCheckBox`), three-state check boxes (`wCheckBox3`), left-text check boxes (`wCheckBoxLT`), and three-state left-text checkboxes (`wCheckBox3LT`).

The non-three-state checkboxes alternate between two states when the user clicks on the check box (or its caption): checked and unchecked. The three-state check boxes alternate between three states: unchecked, checked, and grayed (don't care).

The standard (non-LT) checkboxes draw their check boxes immediately to the left of the caption (that is, the text is to the right of the check box). The LT (left text) check boxes draw their text to the left of the check box.

CheckBox declarations let you specify an "onClick" event handler that HOWL will call whenever the user clicks on a checkbox and changes its state. This argument should either be the name of a `widgetProc` procedure



or NULL (if you don't want HOWL to call any procedure, which is actually a common occurrence with checkboxes).

### 18.2.4.1 wCheckable

```
wCheckable
(
    checkBoxID,      // Identifier for checkbox
    caption,         // Caption for checkbox
    style,           // Style for checkbox
    x,               // x position on form
    y,               // y position on form
    w,               // width
    h,               // height
    onClick          // "on click" event handler (or NULL)
);
```

This declaration creates a generic checkbox on the form. The style argument must be one of the following constants:

w.BS_3STATE	Creates a button that is the same as a check box, except that the box can be grayed as well as checked or unchecked. Use the grayed state to show that the state of the check box is not determined.
w.BS_AUTO3STATE	Creates a button that is the same as a three-state check box, except that the box changes its state when the user selects it. The state cycles through checked, grayed, and unchecked.
w.BS_AUTOCHECKBOX	Creates a button that is the same as a check box, except that the check state automatically toggles between checked and unchecked each time the user selects the check box.
w.BS_CHECKBOX	Creates a small, empty check box with text. By default, the text is displayed to the right of the check box. To display the text to the left of the check box, combine this flag with the w.BS_LEFTTEXT style (or on Windows 95 only, with the equivalent w.BS_RIGHTBUTTON style).

Possibly OR'd logically with one of the following constants (as appropriate):

w.BS_BOTTOM	Places text at the bottom of the button rectangle.
w.BS_LEFTTEXT	Places text on the left side of the radio button or check box when combined with a radio button or check box style. Same as the w.BS_RIGHTBUTTON style.
w.BS_CENTER	Centers text horizontally in the button rectangle.
w.BS_LEFT	Left-justifies the text in the button rectangle. However, if the button is a check box or radio button that does not have the w.BS_RIGHTBUTTON style, the text is left justified on the right side of the check box or radio button.
w.BS_PUSHLIKE	Makes a button (such as a check box, three-state check box, or radio button) look and act like a push button. The button looks raised when it isn't pushed or checked, and sunken when it is pushed or checked. w.BS_RIGHTRight-justifies text in the button rectangle. However, if the button is a check box or radio button that does not have the w.BS_RIGHTBUTTON style, the text is right justified on the right side of the check box or radio button.
w.BS_RIGHTBUTTON	Positions a radio button's circle or a check box's square on the right side of the button rectangle. Same as the w.BS_LEFTTEXT style.
w.BS_TOP	Places text at the top of the button rectangle.
w.BS_VCENTER	Places text in the middle (vertically) of the button rectangle.

### 18.2.4.2 wCheckBox

```
wCheckBox
```

```
(
    checkBoxID,      // Identifier for checkbox
    caption,         // Caption for checkbox
    x,               // x position on form
    y,               // y position on form
    w,               // width
    h,               // height
    onClick          // "on click" event handler (or NULL)
);
```

This declaration creates a standard checkbox on the form.

### 18.2.4.3 wCheckBox3

```
wCheckBox3
(
    checkBoxID,      // Identifier for checkbox
    caption,         // Caption for checkbox
    x,               // x position on form
    y,               // y position on form
    w,               // width
    h,               // height
    onClick          // "on click" event handler (or NULL)
);
```

This declaration creates a three-state checkbox on the form.

### 18.2.4.4 wCheckBox3LT

```
wCheckBox3LT
(
    checkBoxID,      // Identifier for checkbox
    caption,         // Caption for checkbox
    x,               // x position on form
    y,               // y position on form
    w,               // width
    h,               // height
    onClick          // "on click" event handler (or NULL)
);
```

This declaration creates a three-state, left-text, checkbox on the form.

### 18.2.4.5 wCheckBoxLT

```
wCheckBoxLT
(
    checkBoxID,      // Identifier for checkbox
    caption,         // Caption for checkbox
    x,               // x position on form
    y,               // y position on form
    w,               // width
    h,               // height
    onClick          // "on click" event handler (or NULL)
);
```

This declaration creates a left-text checkbox on the form.

## 18.2.5 wComboBox

A combobox is a combination editBox, listBox, and pull-down menu. The user can type text directly into the editBox section of the combo box or click on the arrow on the right side of the widget to open up a pull-down menu from which the user can select an item.

```

wComboBox
(
    comboBoxID,          // ComboBox name (an identifier)
    "combo box",         // Initial string in the edit box (usually and empty string)
    x,                   // x
    y,                   // y
    w,                   // width
    h,                   // height
    sorted,              // true or false
    onCBSelChange,       // onSelChange handler (or NULL)
    "comboBox1",         // List of initial string values for the list
    "comboBox2",         // This list may contain zero or more items.
    "comboBox3"
);

```

The `comboBoxID` argument is the HLA identifier name that HOWL will use for the `comboBox` object within the `wForm` declaration; this name should be unique within the form declaration.

The second argument is a string that HOWL will use as the default value of the edit box field when the form is first created. Most often, this will be the empty string. Note that once the user enters data into the edit box or selects and entry from the pull-down menu list, the initial string value is lost.

The `x`, `y`, `w`, and `h` fields specify the position and size of the combo box on the form.

The `sorted` field is a boolean value that determines whether the fields of the pull-down menu list are sorted or remain in their "inserted" order. If this field is true, then Windows will sort each entry you add to the list (including the initial entries). If this field is false, then Windows leaves the entries in the order that you add them. Note that it is certainly possible to add and delete fields while the program is running; see the discussion of the `wComboBox_t` type later in this document. Most often, you'll probably specify false for this field.

The `onCBSelChange` field lets you specify an "on selection change" event handler or NULL if you don't want HOWL to invoke an event handler when the selection change. Normally, you'll put NULL in this field because you'll normally read the text from the widget when you press some other button or when some other event occurs, not when the user changes the text selection in the edit box or selects some entry from the pull-down menu list.

The remaining entries in the `wComboBox` declaration are optional. These entries, if present, must all be string constants that HOWL will use to populate the pull-down menu list. Note that you can add strings to the list at run time, so you don't have to populate the list at declaration time. However, for many lists you'll know the items the user can select from at design time, so you can fill in those entries in the `wComboBox` declaration.

## 18.2.6 wDragListBox

A `wDragListBox` object is similar to a `wListBox` (list box) object that provides the user with the ability to rearrange items in the list box. The declaration of a `wDragListBox` is

```

wDragListBox
(
    dlName,              // DragListBox name (HLA identifier)
    x,                   // x-coordinate
    y,                   // y-coordinate
    w,                   // width
    h,                   // height
    onListBoxClick,      // onClick handler
    "DragListBox1",      // Initial list population (can be empty)
    "DragListBox2",
    "DragListBox3"
);

```

The `dlName` argument is the name that HOWL will use in the `wForm` declaration for this `wDragListBox` object. This name must be unique within the `wForm` declaration.

The `x`, `y`, `w`, and `h` arguments specify the position and size of the `wDragListBox` object on the form.

The `onListBoxClick` argument is the name of a `widgetProc` that HOWL will call whenever the user clicks on one of the list items. This field can be NULL, in which case HOWL won't bother to call any procedure when the user clicks on an item (this is actually a common situation; usually the program will determine the

currently selected item in a `wDragListBox` when some other event occurs, and ignore the immediate changes that might occur in a `wDragListBox` object).

The remaining objects are optional. If present, they must all be strings and the `wDragListBox` declaration uses these strings to initially populate the `wDragListBox` object.

## 18.2.7 wEditBox

A `wEditBox` object allows the user to enter string data into a program.

```
wEditBox
(
    ebName,      // HLA identifier for this object
    InitialText, // Initial text for edit box
    x,          // x position
    y,          // y position
    w,          // width
    h,          // height
    s,          // style
    onChange    // onChange handler (can be NULL)
);
```

The `ebName` argument is the name that HOWL will use in the `wForm` declaration for this `wEditBox` object. This name must be unique within the `wForm` declaration.

The `InitialText` argument is a string (usually empty) that HOWL uses to initialize the edit box's text field when the form is first created.

The `x`, `y`, `w`, and `h` arguments specify the position and size of the `wEditBox` object on the form.

The `s` argument is the edit box style. This is any of the following edit box styles that HOWL logically ORs with the `w.AUTOHSCROLL` style:

<code>w.ES_AUTOHSCROLL</code>	Automatically scrolls text to the right by 10 characters when the user types a character at the end of the line. When the user presses the ENTER key, the control scrolls all text back to position zero.
<code>w.ES_AUTOVSCROLL</code>	Automatically scrolls text up one page when the user presses the ENTER key on the last line.
<code>w.ES_CENTER</code>	Centers text in a multiline edit control.
<code>w.ES_LEFT</code>	Left-aligns text.
<code>w.ES_LOWERCASE</code>	Converts all characters to lowercase as they are typed into the edit control.
<code>w.ES_MULTILINE</code>	Designates a multiline edit control. The default is single-line edit control.

When the multiline edit control is in a dialog box, the default response to pressing the ENTER key is to activate the default button. To use the ENTER key as a carriage return, use the `ES_WANTRETURN` style.

When the multiline edit control is not in a dialog box and the `ES_AUTOVSCROLL` style is specified, the edit control shows as many lines as possible and scrolls vertically when the user presses the ENTER key. If you do not specify `ES_AUTOVSCROLL`, the edit control shows as many lines as possible and beeps if the user presses the ENTER key when no more lines can be displayed.

If you specify the `ES_AUTOHSCROLL` style, the multiline edit control automatically scrolls horizontally when the caret goes past the right edge of the control. To start a new line, the user must press the ENTER key. If you do not specify `ES_AUTOHSCROLL`, the control automatically wraps words to the beginning of the next line when necessary. A new line is also started if the user presses the ENTER key. The window size determines the position of the word wrap. If the window size changes, the word wrapping position changes and the text is redisplayed.

Multiline edit controls can have scroll bars. An edit control with scroll bars processes its own scroll bar messages. Note that edit controls without scroll bars

scroll as described in the previous paragraphs and process any scroll messages sent by the parent window.

<code>w.ES_NOHIDSESEL</code>	Negates the default behavior for an edit control. The default behavior hides the selection when the control loses the input focus and inverts the selection when the control receives the input focus. If you specify <code>ES_NOHIDSESEL</code> , the selected text is inverted, even if the control does not have the focus.
<code>w.ES_NUMBER</code>	Allows only digits to be entered into the edit control.
<code>w.ES_OEMCONVERT</code>	Converts text entered in the edit control. The text is converted from the Windows character set to the OEM character set and then back to the Windows set. This ensures proper character conversion when the application calls the <code>CharToOem</code> function to convert a Windows string in the edit control to OEM characters. This style is most useful for edit controls that contain filenames.
<code>w.ES_PASSWORD</code>	Displays an asterisk (*) for each character typed into the edit control. You can use the <code>EM_SETPASSWORDCHAR</code> message to change the character that is displayed.
<code>w.ES_READONLY</code>	Prevents the user from typing or editing text in the edit control.
<code>w.ES_RIGHT</code>	Right-aligns text in a multiline edit control.
<code>w.ES_UPPERCASE</code>	Converts all characters to uppercase as they are typed into the edit control.
<code>w.ES_WANTRETURN</code>	Specifies that a carriage return be inserted when the user presses the ENTER key while entering text into a multiline edit control in a dialog box. If you do not specify this style, pressing the ENTER key has the same effect as pressing the dialog box's default push button. This style has no effect on a single-line edit control.

The `onChange` argument is the name of a `widgetProc` that HOWL will call whenever the user changes any text in the edit box. This field can be `NULL`, in which case HOWL won't bother to call any procedure when the user changes the text (this is actually a common situation; usually the program will determine the currently selected item in a `wEditBox` when some other event occurs, and ignore the immediate changes that might occur in a `wEditBox` object). Note that if `onChange` is non-`NULL`, then HOWL will call the `widgetProc` any time there is a single-character change to the edit box; this is probably more often than you'd like, which is why this field is generally `NULL` and applications simply read the data from the edit box when some other event occurs.

## 18.2.8 wEllipse

```
wEllipse
(
    ellipseName,    // HLA identifier
    x,              // x
    y,              // y
    w,              // width
    h,              // height
    lineColor,      // linecolor (RGB)
    fillColor,      // Ellipse interior color (RGB)
    bgColor         // Ellipse exterior color (RGB)
)
```

The `ellipseName` argument is the identifier name that HOWL uses in the `wForm` declaration for the ellipse object. This name must be unique within the `wForm` declaration.

The `x`, `y`, `w`, and `h` fields specify the coordinates and sizes for the bounding box surrounding the ellipse. If these coordinates and sizes form a square, then you'll draw a circle on the form.

The `lineColor` argument specifies an RGB value that HOWL uses when drawing the outline of the ellipse (the pen color). `wEllipse` objects always draw the outline with a solid line.

The `fillColor` argument specifies an RGB value that HOWL uses to fill the interior of the ellipse. `wEllipse` objects always fill the interior with a solid brush (color).

The `bkgColor` argument specifies an RGB value that HOWL uses to fill the rectangular area described by `x`, `y`, `w`, and `h` that is outside the ellipse. As a general rule, this should be the same color as the background color for the form unless you want a visible rectangle surrounding the ellipse. If the H.O. byte of this color value contains \$FF, then HOWL will use the RGB value in the L.O. three bytes to fill the initial object window's background color but will not redraw the background thereafter (making the object's background transparent).

## 18.2.9 wIcon

The `wIcon` declaration places an icon object on the form.

```
wIcon
(
    iconIdentifier,          // icon name (HLA identifier)
    IconResourceStr,        // icon resource value
    x,                      // x
    y,                      // y
    w,                      // width
    h,                      // height
    bkgColor                // background color
)
```

The `iconIdentifier` field is an HLA identifier that HOWL uses in the form declaration for this particular icon. This name must be unique within the `wForm` declaration.

The `IconResourceStr` argument is either a string containing the name of an internal icon resource value or a constant that is less than \$1\_0000 (that specifies a system icon). If this is a string, it is not a filename for the icon, rather it is a resource ID produced by a resource compiler. See the HLA examples (HOWL directory) for examples that show how to use the resource compiler to produce icons for a program. If this is a value less than \$1\_0000, then it must be one of the following values:

- `w.IDI_APPLICATION`
- `w.IDI_ASTERISK`
- `w.IDI_EXCLAMATION`
- `w.IDI_HAND`
- `w.IDI_QUESTION`
- `w.IDI_WINLOGO`

Because of syntactical issues with the HLA macro language, if you want to specify these constants as the `IconResourceStr` argument (which normally must be a string), the best way to do this is to use instruction composition thusly:

```
wIcon
(
    icon1,                  // icon name
    mov( w.IDI_APPLICATION, eax), // icon resource value
    10,                    // x
    440,                   // y
    32,                    // width
    32,                    // height
    bkgColor_g             // background color
)
```

The `x`, `y`, `w`, and `h` fields specify the coordinates and sizes for the bounding box surrounding the icon. If this bounding box is too small for the icon, portions of the icon will be clipped. If this bounding box is too big for the icon, then HOWL will fill the extra area with the background color.

The `bkgColor` argument specifies the background color that HOWL will use to fill the bounding box for the icon if the icon itself is smaller than the bounding box specified by the `x`, `y`, `w`, and `h` fields.

## 18.2.10 wGroupBox..endwGroupBox

A `wGroupBox` object is a container. It draws a rectangular box on a form that contains other objects. All the widgets you declare between a `wGroupBox` statement and the corresponding `endwGroupBox` terminator will be contained by the group box (and can be treated as a whole) at run time.

Note that placing `wRadioButtons` within a `wGroupBox` object does not automatically create a set of radio set buttons. See the `wRadioSet` declaration for that purpose. `wGroupBox` objects really exist just to make the form look pretty.

```

wGroupBox
(
    groupBoxID,          // HLA identifier
    Caption,             // String caption for group box
    x,                   // x position
    y,                   // y position
    w,                   // width
    h,                   // height
)

    <<Other widget declarations, not including Radio Sets >>

endwGroupBox

```

## 18.2.11 wLabel

```

wLabel
(
    labelID,             // HLA identifier
    labelString,         // Label string
    x,                   // x
    y,                   // y
    w,                   // width
    h,                   // height
    style,               // Alignment and style
    textColor,           // Foreground color
    bkgColor             // Background color
);

```

The `wLabel` declaration lets you place a text string on the form.

The `labelID` argument is the name of the `wLabel` field within the form's class. This name must be unique within the `wForm` declaration.

The `x`, `y`, `w`, and `h` fields specify the coordinates and sizes for the bounding box surrounding the label's string. If this bounding box is too small for the string, portions of the string will be clipped. If this bounding box is too big for the icon, then HOWL will fill the extra area with the background color.

The `style` field is one or more of the following Windows constants logically-ORed together:

<code>w.DT_BOTTOM</code>	Bottom-justifies text. This value must be combined with <code>DT_SINGLELINE</code> .
<code>w.DT_CENTER</code>	Centers text horizontally.
<code>w.DT_EXPANDTABS</code>	Expands tab characters. The default number of characters per tab is eight.
<code>w.DT_LEFT</code>	Aligns text to the left.
<code>w.DT_NOPREFIX</code>	Turns off processing of prefix characters. Normally, <code>DrawText</code> interprets the mnemonic-prefix character <code>&amp;</code> as a directive to underscore the character that follows, and the mnemonic-prefix characters <code>&amp;&amp;</code> as a directive to print a single <code>&amp;</code> . By specifying <code>DT_NOPREFIX</code> , this processing is turned off.
<code>w.DT_RIGHT</code>	Aligns text to the right.
<code>w.DT_SINGLELINE</code>	Displays text on a single line only. Carriage returns and linefeeds do not break the line.
<code>w.DT_TOP</code>	Top-justifies text (single line only).
<code>w.DT_VCENTER</code>	Centers text vertically (single line only).
<code>w.DT_WORDBREAK</code>	Breaks words. Lines are automatically broken between words if a word would extend past the edge of the rectangle specified by the <code>lpRect</code> parameter. A carriage return-linefeed sequence also breaks the line.

For example, to vertically and horizontally center a string within the `wLabel` bounding box, you would use the following constant for the `style` field:

w.DT\_CENTER | w.DT\_VCENTER | w.DT\_SINGLELINE

The `textColor` and `bkgColor` fields are RGB values that specify the (solid) colors used to draw the text and the background for the text. Usually the text's background color is the same as the form's background color unless you are trying to create a special effect.

## 18.2.12 wListBox

A `wListBox` object contains a sequence of strings from which the user can select a single entry. The declaration of a `wListBox` is

```
wListBox
(
    lbName,          // ListBox name (HLA identifier)
    x,               // x-coordinate
    y,               // y-coordinate
    w,               // width
    h,               // height
    sort,            // true or false
    onListBoxClick,  // onClick handler
    "ListBox1",      // Initial list population (can be empty)
    "ListBox2",
    "ListBox3"
);
```

The `lbName` argument is the name that HOWL will use in the `wForm` declaration for this `wListBox` object. This name must be unique within the `wForm` declaration.

The `x`, `y`, `w`, and `h` arguments specify the position and size of the `wListBox` object on the form.

The `onListBoxClick` argument is the name of a `widgetProc` that HOWL will call whenever the user clicks on one of the list items. This field can be `NULL`, in which case HOWL won't bother to call any procedure when the user clicks on an item (this is actually a common situation; usually the program will determine the currently selected item in a `wListBox` when some other event occurs).

The remaining objects are optional. If present, they must all be strings and the `wListBox` declaration uses these strings to initially populate the `wListBox` object.

## 18.2.13 wPasswdBox

A `wPasswdBox` object is very similar to a `wEditBox` object insofar as it allows the user to enter a string of text onto the form. The difference is that the `wPasswdBox` object displays asterisks (or some other character) when the user types a string into the editbox. This shields sensitive information from prying eyes.

```
wPasswdBox
(
    pbName,          // HLA identifier for this object
    InitialText,     // Initial text for edit box
    x,               // x position
    y,               // y position
    w,               // width
    h,               // height
    s,               // style
    onChange         // onChange handler (can be NULL)
);
```

The `pbName` argument is the name that HOWL will use in the `wForm` declaration for this `wPasswdBox` object. This name must be unique within the `wForm` declaration.

The `InitialText` argument is a string (usually empty) that HOWL uses to initialize the password box's text field when the form is first created. This is almost always the empty string.



The `x`, `y`, `w`, and `h` arguments specify the position and size of the `wPasswdBox` object on the form.

The `s` parameter is the Windows edit box style that HOWL logically ORs with the value `(w.ES_AUTOHSCROLL | w.ES_PASSWORD)`. See the discussion of the available styles in the description of the `wEditBox` object.

The `onChange` argument is the name of a `widgetProc` that HOWL will call whenever the user changes any text in the password box. This field can be `NULL`, in which case HOWL won't bother to call any procedure when the user changes the text (this is actually a common situation; usually the program will determine the currently selected item in a `wPasswdBox` when some other event occurs, and ignore the immediate changes that might occur in a `wPasswdBox` object). Note that if `onChange` is non-`NULL`, then HOWL will call the `widgetProc` any time there is a single-character change to the password box; this is probably more often than you'd like, which is why this field is generally `NULL` and applications simply read the data from the password box when some other event occurs.

## 18.2.14 wPie

The `wPie` declaration defines a graphic object on the form that is a "pie slice", that is, a portion of a pie graph.

```
wPie
(
    pieName,          // HLA identifier
    x,                // x
    y,                // y
    w,                // width
    h,                // height
    startAngle,       // Starting handle (in degrees)
    endAngle,         // Ending angle (in degrees)
    lineColor,        // linecolor (RGB)
    fillColor,        // Ellipse interior color (RGB)
    bkgColor          // Ellipse exterior color (RGB)
)
```

The `pieName` argument is the identifier name that HOWL uses in the `wForm` declaration for the pie slice object. This name must be unique within the `wForm` declaration.

The `x`, `y`, `w`, and `h` fields specify the coordinates and sizes for the bounding box surrounding the pie slice.

The `startAngle` parameter is a `real64` value that specifies the starting angle of the pie slice. The angle is specified in degrees. Angles are measured in a counter-clockwise fashion from the vertical line going from the middle of the bounding box to the top of the bounding box (warning: this is not intuitive).

The `endAngle` parameter is a `real64` value that specifies the ending angle of the pie slice. The angle is specified in degrees. The `wPie` object draws a slice of a pie graph filling in the ellipse from the `startAngle` to the `endAngle` in a counter-clockwise fashion.

The `lineColor` argument specifies an RGB value that HOWL uses when drawing the outline of the pie slice (the pen color). `wPie` objects always draw the outline with a solid line.

The `fillColor` argument specifies an RGB value that HOWL uses to fill the interior of the pie slice. `wPie` objects always fill the interior with a solid brush (color).

The `bkgColor` argument specifies an RGB value that HOWL uses to fill the rectangular area described by `x`, `y`, `w`, and `h` that is outside the pie slice. As a general rule, this should be the same color as the background color for the form unless you want a visible rectangle surrounding the pie slice. If the H.O. byte of this color value contains `$FF`, then HOWL will use the RGB value in the L.O. three bytes to fill the initial object window's background color but will not redraw the background thereafter (making the object's background transparent).

## 18.2.15 wPolygon

The `wPolygon` object draws a multi-vertex polygon on the screen.

```
wPolygon
(
    polyName,        // HLA identifier
    x,                // x
    y,                // y
    w,                // width
```

```

    h,                // height
    lineColor,        // linecolor (RGB)
    fillColor,        // Ellipse interior color (RGB)
    bkgColor,         // Ellipse exterior color (RGB)
    x1,               // Optional points list
    y1,               // Must have an even number of coordinates
    x2,
    y2,
    .
    .
    .
    xn,
    yn
)

```

The `polyName` argument is the identifier name that HOWL uses in the `wForm` declaration for the polygon object. This name must be unique within the `wForm` declaration.

The `x`, `y`, `w`, and `h` fields specify the coordinates and sizes for the bounding box surrounding the polygon.

The `lineColor` argument specifies an RGB value that HOWL uses when drawing the outline of the polygon (the pen color). `wPolygon` objects always draw the outline with a solid line.

The `fillColor` argument specifies an RGB value that HOWL uses to fill the interior of the polygon. `wPolygon` objects always fill the interior with a solid brush (color).

The `bkgColor` argument specifies an RGB value that HOWL uses to fill the rectangular area described by `x`, `y`, `w`, and `h` that is outside the polygon. As a general rule, this should be the same color as the background color for the form unless you want a visible rectangle surrounding the polygon. If the H.O. byte of this color value contains \$FF, then HOWL will use the RGB value in the L.O. three bytes to fill the initial object window's background color but will not redraw the background thereafter (making the object's background transparent).

The remain arguments always appear in pairs and specify the points that make up the polygon. If you specify `n` points (`n*2` arguments), HOWL will draw `n` lines between each pair of points (and between (`xn,yn`) and (`x1,y1`) to complete the closed polygon).

## 18.2.16 wBitmap

The `wBitmap` declaration creates a bitmapped object on the form.

```

wBitmap
(
    bmName,           // HLA identifier
    bmResource,       // Bitmap resource name
    x,                 // x
    y,                 // y
    w,                 // width
    h,                 // height
    bkgColor           // RGB background color
)

```

The `bmName` argument is the identifier name that HOWL uses in the `wForm` declaration for the bitmapped object. This name must be unique within the `wForm` declaration.

The `bmResource` argument is a string constant specifying the name of the bitmap resource within the executable file. Note that this is not a filename on the disk. You must use the resource editor to compile a bitmap file into the executable file. The name you provide to the resource editor for this bitmapped object is the name you use for the `bmResource` string.

The `x`, `y`, `w`, and `h` fields specify the coordinates and sizes for the bounding box surrounding the bitmap. If the `w` and `h` fields are too small, Windows will truncate the bitmap when it draws it. If the `w` and `h` fields are larger than the bitmap, Windows will fill the extra area with the value of the `bkgColor` argument.

The `bkgColor` argument specifies an RGB value that HOWL uses to fill the rectangular area described by `x`, `y`, `w`, and `h` that is outside the bitmap. As a general rule, this should be the same color as the background color for the form unless you want a visible rectangle surrounding the bitmap. If the H.O. byte of this color value contains \$FF, then HOWL will use the RGB value in the L.O. three bytes to fill the initial object window's background color but will not redraw the background thereafter (making the object's background transparent).

## 18.2.17 wProgressBar

The `wProgressBar` declaration creates a progress bar object on the form.

```
wProgressBar
(
    pbName,          // HLA identifier
    x,               // x
    y,               // y
    w,               // width
    h                // height
)
```

The `pbName` argument is the identifier name that HOWL uses in the `wForm` declaration for the progress bar object. This name must be unique within the `wForm` declaration.

The `x`, `y`, `w`, and `h` fields specify the coordinates and sizes for the bounding box surrounding the progress bar.

See the description of the `wProgressBar_t` class type later in the document to learn how to specify the current progress in the progress bar.

## 18.2.18 wPushButton

```
wPushButton
(
    pbID,            // Identifier for push button
    caption,         // Caption for push button
    x,               // x position on form
    y,               // y position on form
    w,               // width
    h,               // height
    onClick          // "on click" event handler (or NULL)
);
```

The `pbName` argument is the identifier name that HOWL uses in the `wForm` declaration for the push button object. This name must be unique within the `wForm` declaration.

The `caption` argument is a string that Windows will display on the push button.

The `x`, `y`, `w`, and `h` fields specify the coordinates and sizes for the bounding box surrounding the push button.

The `onClick` argument is either `NULL` or the name of a `widgetProc` that HOWL will call when the user presses the corresponding button on the form.

## 18.2.19 Radio Button Objects

`wForm` declarations allow you to place radio buttons directly on a form or you can group a set of radio buttons together in a radio button set. In general, you'll rarely use the first form because `wRadioButton` and `wRadioButtonLT` objects don't readily exhibit button semantics. On a form by themselves, radio buttons behave just like check box objects so you're better off using check box objects than radio buttons for this purpose. Generally, `wRadioButton` and `wRadioButtonLT` objects are useful when you're building a form dynamically at run time rather than at design time with the HOWL declarative language. Nevertheless, the HOWL declarative language includes entries for `wRadioButton` and `wRadioButtonLT` objects for the sake of completeness.

### 18.2.19.1 wRadioButton

```
wRadioButton
(
    rbID,            // Identifier for radio button
    caption,         // Caption for radio button
    style,           // Style for radio button
    x,               // x position on form
    y,               // y position on form
)
```

```

    w,           // width
    h,           // height
    onClick      // "on click" event handler (or NULL)
);

```

The `rbName` argument is the identifier name that HOWL uses in the `wForm` declaration for the radio button object. This name must be unique within the `wForm` declaration.

The `caption` argument is a string that Windows will display to the right of the radio button.

The `style` argument is either 0 (for standalone radio buttons) or one of the following constants (for radio button groups):

For the first radio button in a group:

```
w.BS_AUTORADIOBUTTON | w.WS_GROUP | w.WS_TABSTOP
```

For all but the first radio button in a group:

```
w.BS_AUTORADIOBUTTON
```

Note that you should really use the `wRadioSet` object to create sets of radio buttons rather than grouping them manually. Note that you must declare all grouped radio buttons consecutively in your source file. Any intervening widgets will end a radio set button group.

The `x`, `y`, `w`, and `h` fields specify the coordinates and sizes for the bounding box surrounding the radio button.

The `onClick` argument is either `NULL` or the name of a widgetProc that HOWL will call when the user presses the corresponding radio button on the form.

## 18.2.19.2 wRadioButtonLT

```

wRadioButtonLT
(
    rbID,           // Identifier for radio button
    caption,        // Caption for radio button
    style,          // Style for radio button
    x,              // x position on form
    y,              // y position on form
    w,              // width
    h,              // height
    onClick         // "on click" event handler (or NULL)
);

```

`wRadioButtonLT` objects are identical to `wRadioButton` objects except the caption text is drawn on the left side of the radio button rather than on the right side.

## 18.2.19.3 wRadioSet..endwRadioSet

Functional radio buttons are created as part of a radio button set. The `wRadioSet..endwRadioSet` block encapsulates a set of `wRadioSetButton` and `wRadioSetButtonLT` objects that HOWL treats as a single set of radio buttons rather than independent buttons. The `wRadioSet` declaration takes the following form:

```

wRadioSet
(
    rsID,           // Identifier for radio set
    caption,        // Caption for radio set group box
    x,              // x position on form
    y,              // y position on form
    w,              // width
    h,              // height
);

<< radio set button declarations >>

```

```
endwRadioSet
```

Only `wRadioSetButton` and `wRadioSetButtonLT` declarations may appear within a `wRadioSet..endwRadioSet` declaration and you cannot nest `wRadioSet..endwRadioSet` declarations. The `wRadioSet..endwRadioSet` declaration creates a group box with the specified bounding box. It draws the caption string through the line of the bounding box in the upper left hand corner.

All radio set buttons appearing in a `wRadioSet` group are treated as a single set of radio buttons. At most one radio set button will be checked in the group; pressing one button unchecks any other buttons in the same group.

Note that a `wRadioSet` object is a container object. It contains all the radio set buttons associated with the radio set.

### 18.2.19.3.1 wRadioSetButton

```
wRadioSetButton
(
    rsbtnID,          // HLA identifier
    caption,          // String caption for radio button
    x,                // x position
    y,                // y position
    w,                // width
    h,                // height
    onClick           // "on click" event handler
);
```

The `wRadioSetButton` declaration may only appear within a `wRadioSet..endwRadioSet` statement.

The `rsbtnID` argument must be a unique (to the form) HLA identifier. HOWL uses this identifier as the field name within the form class of the `wForm..endwForm` declaration.

The `caption` field is a string that HOWL displays to the right of the radio button image.

The `x`, `y`, `w`, and `h` fields specify the coordinates and sizes for the bounding box surrounding the radio button.

The `onClick` argument is either `NULL` or the name of a `widgetProc` that HOWL will call when the user presses the corresponding radio button on the form.

### 18.2.19.3.2 wRadioSetButtonLT

```
wRadioSetButtonLT
(
    rsbtnID,          // HLA identifier
    caption,          // String caption for radio button
    x,                // x position
    y,                // y position
    w,                // width
    h,                // height
    onClick           // "on click" event handler
);
```

The `wRadioSetButtonLT` declaration may only appear within a `wRadioSet..endwRadioSet` statement.

The `rsbtnID` argument must be a unique (to the form) HLA identifier. HOWL uses this identifier as the field name within the form class of the `wForm..endwForm` declaration.

The `caption` field is a string that HOWL displays to the left of the radio button image.

The `x`, `y`, `w`, and `h` fields specify the coordinates and sizes for the bounding box surrounding the radio button.

The `onClick` argument is either `NULL` or the name of a `widgetProc` that HOWL will call when the user presses the corresponding radio button on the form.

## 18.2.20 wRectangle

```

wRectangle
(
    rectName,          // HLA identifier
    x,                  // x
    y,                  // y
    w,                  // width
    h,                  // height
    lineColor,          // linecolor (RGB)
    fillColor           // Rectangle interior color (RGB)
)

```

The `rectName` argument is the identifier name that HOWL uses in the `wForm` declaration for the rectangle object. This name must be unique within the `wForm` declaration.

The `x`, `y`, `w`, and `h` fields specify the coordinates and sizes for the bounding box surrounding the rectangle.

The `lineColor` argument specifies an RGB value that HOWL uses when drawing the outline of the rectangle (the pen color). `wRectangle` objects always draw the outline with a solid line.

The `fillColor` argument specifies an RGB value that HOWL uses to fill the interior of the rectangle. `wRectangle` objects always fill the interior with a solid brush (color).

Note that there is no background color (as exists for other graphic objects). This is because the rectangle completely fills the bounding box so there is no need to fill the background area as it never shows through.

## 18.2.21 wRoundRect

```

wRoundRect
(
    rrectName,          // HLA identifier
    x,                  // x
    y,                  // y
    w,                  // width
    h,                  // height
    cw,                 // Corner width
    cht,                // Corner height
    lineColor,          // linecolor (RGB)
    fillColor,          // Round rectangle interior color (RGB)
    BkgColor            // Round rectangle exterior color (RGB)
)

```

The `rrectName` argument is the identifier name that HOWL uses in the `wForm` declaration for the round rectangle object. This name must be unique within the `wForm` declaration.

The `x`, `y`, `w`, and `h` fields specify the coordinates and sizes for the bounding box surrounding the round rectangle.

The `cw` and `cht` arguments specify the width and height of the ellipse used to draw the corners of the round rectangle.

The `lineColor` argument specifies an RGB value that HOWL uses when drawing the outline of the round rectangle (the pen color). `wRoundRect` objects always draw the outline with a solid line.

The `fillColor` argument specifies an RGB value that HOWL uses to fill the interior of the rectangle. `wRectangle` objects always fill the interior with a solid brush (color).

The `bkgColor` argument specifies an RGB value that HOWL uses to fill the rectangular area described by `x`, `y`, `w`, and `h` that is outside the round rectangle. As a general rule, this should be the same color as the background color for the form unless you want a visible rectangle surrounding the bitmap. If the H.O. byte of this color value contains \$FF, then HOWL will use the RGB value in the L.O. three bytes to fill the initial object window's background color but will not redraw the background thereafter (making the object's background transparent).

## 18.2.22 wScrollBar

```
wScrollBar
(
    scrollBarID,    // Scrollbar name (HLA id)
    x,              // x
    y,              // y
    w,              // width
    h,              // height
    style,          // Scroll bar style
    onChange        // On change handler
)
```

The `scrollBarID` argument is the identifier name that HOWL uses in the `wForm` declaration for the scroll bar object. This name must be unique within the `wForm` declaration.

The `x`, `y`, `w`, and `h` fields specify the coordinates and sizes for the bounding box surrounding the scroll bar.

The `style` argument specifies the scroll bar style and is the logical-OR of zero or more of the following constants:

<code>w.SBS_BOTTOMALIGN</code>	Aligns the bottom edge of the scroll bar with the bottom edge of the rectangle defined by the <code>CreateWindow</code> parameters <code>x</code> , <code>y</code> , <code>nWidth</code> , and <code>nHeight</code> . The scroll bar has the default height for system scroll bars. Use this style with the <code>SBS_HORZ</code> style.
<code>w.SBS_HORZ</code>	Designates a horizontal scroll bar. If neither the <code>SBS_BOTTOMALIGN</code> nor <code>SBS_TOPALIGN</code> style is specified, the scroll bar has the height, width, and position specified by the parameters of <code>CreateWindow</code> .
<code>w.SBS_LEFTALIGN</code>	Aligns the left edge of the scroll bar with the left edge of the rectangle defined by the parameters of <code>CreateWindow</code> . The scroll bar has the default width for system scroll bars. Use this style with the <code>SBS_VERT</code> style.
<code>w.SBS_RIGHTALIGN</code>	Aligns the right edge of the scroll bar with the right edge of the rectangle defined by the parameters of <code>CreateWindow</code> . The scroll bar has the default width for system scroll bars. Use this style with the <code>SBS_VERT</code> style.
<code>w.SBS_SIZEBOX</code>	Designates a size box. If you specify neither the <code>SBS_SIZEBOXBOTTOMRIGHTALIGN</code> nor the <code>SBS_SIZEBOXTOPLEFTALIGN</code> style, the size box has the height, width, and position specified by the parameters of <code>CreateWindow</code> .
<code>w.SBS_SIZEBOXBOTTOMRIGHTALIGN</code>	Aligns the lower-right corner of the size box with the lower-right corner of the rectangle specified by the parameters of <code>CreateWindow</code> . The size box has the default size for system size boxes. Use this style with the <code>SBS_SIZEBOX</code> style.
<code>w.SBS_SIZEBOXTOPLEFTALIGN</code>	Aligns the upper-left corner of the size box with the upper-left corner of the rectangle specified by the parameters of <code>CreateWindow</code> . The size box has the default size for system size boxes. Use this style with the <code>SBS_SIZEBOX</code> style.
<code>w.SBS_SIZEGRIP</code>	Same as <code>SBS_SIZEBOX</code> , but with a raised edge.
<code>w.SBS_TOPALIGN</code>	Aligns the top edge of the scroll bar with the top edge of the rectangle defined by the parameters of <code>CreateWindow</code> . The scroll bar has the default height for system scroll bars. Use this style with the <code>SBS_HORZ</code> style.
<code>w.SBS_VERT</code>	Designates a vertical scroll bar. If you specify neither the <code>SBS_RIGHTALIGN</code> nor the <code>SBS_LEFTALIGN</code> style, the scroll bar has the height, width, and position specified by the parameters of <code>CreateWindow</code> .

The `onChange` argument is the name of a `widgetProc` that HOWL will call whenever there is a change made to the scroll bar's position.

## 18.2.23 wTextEdit

A wTextEdit object allows the user to enter a text file object.

```
wTextEdit
(
    teName,          // HLA identifier for this object
    InitialText,     // Initial text for text edit object
    x,               // x position
    y,               // y position
    w,               // width
    h,               // height
    s,               // style
    onChange         // onChange handler (can be NULL)
);
```

The `teName` argument is the name that HOWL will use in the `wForm` declaration for this `wTextEdit` object. This name must be unique within the `wForm` declaration.

The `InitialText` argument is a string (usually empty) that HOWL uses to initialize the text editor's text field when the form is first created.

The `x`, `y`, `w`, and `h` arguments specify the position and size of the `wTextEdit` object on the form.

The `s` argument is the Windows editbox style that HOWL logically ORs with the value (`w.ES_MULTILINE` | `w.ES_WANTRETURN` | `w.WS_HSCROLL` | `w.WS_VSCROLL`). See the discussion of the legal values in the section on `wEditBox`.

The `onChange` argument is the name of a `widgetProc` that HOWL will call whenever the user changes any text in the text editor. This field can be `NULL`, in which case HOWL won't bother to call any procedure when the user changes the text (this is actually the most common situation; usually the program will determine the currently selected item in a `wTextEdit` when some other event occurs, and ignore the immediate changes that might occur in a `wTextEdit` object). Note that if `onChange` is non-`NULL`, then HOWL will call the `widgetProc` any time there is a single-character change to the text edit object; this is probably more often than you'd like, which is why this field is generally `NULL` and applications simply read the data from the text edit object when some other event occurs.

## 18.2.24 wTrackBar

```
wTrackBar
(
    trackBarID,      // Trackbar name (HLA id)
    x,               // x
    y,               // y
    w,               // width
    h,               // height
    style,           // Track bar style
    onChange         // On change handler
);
```

The `trackBarID` argument is the identifier name that HOWL uses in the `wForm` declaration for the track bar object. This name must be unique within the `wForm` declaration.

The `x`, `y`, `w`, and `h` fields specify the coordinates and sizes for the bounding box surrounding the track bar.

The `style` argument specifies the track bar style and is the logical-OR of zero or more of the following constants:

<code>w.TBS_HORZ</code>	Designates a horizontal track bar (this is the default).
<code>w.TBS_TOP</code>	Display tick marks on the top of a horizontal track bar.
<code>w.TBS_BOTTOM</code>	Display tick marks on the bottom of a horizontal track bar (default).
<code>w.TBS_VERT</code>	Designates a vertical track bar.
<code>w.TBS_LEFT</code>	Display tick marks on the left side of a vertical track bar.
<code>w.TBS_RIGHT</code>	Display tick marks on the right side of a vertical track bar (default).



w.TBS\_BOTH                      Display tick marks on both sides of a track bar (vert or horz).

Note that all wTrackBar objects have the w.TBS\_AUTOTICKS style.

The `onChange` argument is the name of a widgetProc that HOWL will call whenever there is a change made to the scroll bar's position.

## 18.2.25 wUpDown

```
wUpDown
(
    upDownID,          // Up/down control object name
    style,              // No special format/style/alignment
    x,                  // x
    y,                  // y
    w,                  // width
    h,                  // height
    min,                // Minimum position
    max,                // Maximum position
    initial,            // Initial position
    onUpDown            // Click event handler
);
```

A wUpDown widget is a pair of arrow buttons that the user can click on to increment or decrement a value. wUpDown objects are stand-alone (see wUpDownEditBox for a version that is connected to an edit box).

The `upDownID` field is an HLA identifier that HOWL uses as the name of the object on the form. This name must be unique within the form class declaration.

The `style` argument is one of the following values:

UDS_ALIGNLEFT	Positions the up-down control next to the left edge of the buddy window. The buddy window is moved to the right and its width decreased to accommodate the width of the up-down control. This style is generally used only with the UDS_AUTOBUDDY style. See wUpDownEditBox for additional details concerning buddy controls.
UDS_ALIGNRIGHT	Positions the up-down control next to the right edge of the buddy window. The width of the buddy window is decreased to accommodate the width of the up-down control. This style is generally used only with the UDS_AUTOBUDDY style. See wUpDownEditBox for additional details concerning buddy controls.
UDS_ARROWKEYS	Causes the up-down control to increment and decrement the position when the UP ARROW and DOWN ARROW keys are pressed.
UDS_AUTOBUDDY	Automatically selects the previous window in the Z order as the up-down control's buddy window.
UDS_HORZ	Causes the up-down control's arrows to point left and right instead of up and down.
UDS_NOTHOUSANDS	Does not insert a thousands separator between every three decimal digits.
UDS_SETBUDDYINT	Causes the up-down control to set the text of the buddy window (using the WM_SETTEXT message) when the position changes. The text consists of the position formatted as a decimal or hexadecimal string.
UDS_WRAP	Causes the position to "wrap" if it is incremented or decremented beyond the ending or beginning of the range.

The `x`, `y`, `w`, and `h` arguments specify the position and size of the wUpDown object on the form. HOWL ignores these values if you specify a non-NULL buddy value; in that case, HOWL uses the bounding box of the wEditBox object to control the placement of the up/down arrows.

The `min` argument specifies the minimum value that a `wUpDown` object will return. If the control's current value is equal to the `min` value and the user presses the down arrow, the `wUpDown` object will not decrement the value.

The `max` argument specifies the maximum value that a `wUpDown` object will return. If the control's current value is equal to the `max` value and the user presses the up arrow, the `wUpDown` object will not increment the value.

The `initial` argument specifies the initial value of the `wUpDown` object when the form is created.

The `onUpDown` argument is either `NULL` or specifies the name of a `widgetProc` procedure that HOWL will call whenever the user presses an up or down arrow on the control. If this field is `NULL`, then HOWL will not call a function whenever an up or down arrow is pressed and the application will have to call an appropriate `wUpDown` method to retrieve the current value of the `wUpDown` control.

## 18.2.26 wUpDownEditBox

```
wUpDownEditBox
(
    upDownID,          // Up/down control object name
    style,              // No special format/style/alignment
    x,                  // x
    y,                  // y
    w,                  // width
    h,                  // height
    min,                // Minimum position
    max,                // Maximum position
    initial,            // Initial position
    onTextChange,       // On Change event handler (edit box)
    onUpDown             // Click event handler (up/down arrow)
);
```

A `wUpDown` widget is a pair of arrow buttons that the user can click on to increment or decrement a value. `wUpDown` objects can be stand-alone or they can be associated with a `wEditBox` object (the "buddy"). When a `wUpDown` object is associated with a buddy `wEditBox` object, the arrows are connect to the edit box and clicking on the up or down arrows produces a string in the `wEditBox` object representing the current value of the `wUpDown` object.

The `style` argument is one of the following values:

<code>UDS_ALIGNLEFT</code>	Positions the up-down control next to the left edge of the buddy window. The buddy window is moved to the right and its width decreased to accommodate the width of the up-down control.
<code>UDS_ALIGNRIGHT</code>	Positions the up-down control next to the right edge of the buddy window. The width of the buddy window is decreased to accommodate the width of the up-down control.
<code>UDS_ARROWKEYS</code>	Causes the up-down control to increment and decrement the position when the UP ARROW and DOWN ARROW keys are pressed.
<code>UDS_AUTOBUDDY</code>	Automatically selects the previous window in the Z order as the up-down control's buddy window. This style shouldn't be used with <code>wUpDownEditBox</code> objects.
<code>UDS_HORZ</code>	Causes the up-down control's arrows to point left and right instead of up and down.
<code>UDS_NOTHOUSANDS</code>	Does not insert a thousands separator between every three decimal digits.
<code>UDS_SETBUDDYINT</code>	Causes the up-down control to set the text of the buddy window (using the <code>WM_SETTEXT</code> message) when the position changes. The text consists of the position formatted as a decimal or hexadecimal string.
<code>UDS_WRAP</code>	Causes the position to "wrap" if it is incremented or decremented beyond the ending or beginning of the range.

The `x`, `y`, `w`, and `h` arguments specify the position and size of the `wUpDown` object on the form. HOWL ignores these values if you specify a non-NULL buddy value; in that case, HOWL uses the bounding box of the `wEditBox` object to control the placement of the up/down arrows.

The `min` argument specifies the minimum value that a `wUpDown` object will return. If the control's current value is equal to the `min` value and the user presses the down arrow, the `wUpDown` object will not decrement the value.

The `max` argument specifies the maximum value that a `wUpDown` object will return. If the control's current value is equal to the `max` value and the user presses the up arrow, the `wUpDown` object will not increment the value.

The `initial` argument specifies the initial value of the `wUpDown` object when the form is created.

The `onTextChanged` argument is either NULL or specifies the name of a `widgetProc` procedure that HOWL will call whenever the user changes a value in the edit box control. If this field is NULL, then HOWL will not call a function whenever the edit box changes and the application will have to call an appropriate `wUpDownEditBox` method to retrieve the current value of the `wUpDownEditBox`'s edit box control. Note that pressing an up or down error will cause a change to the edit box, which will call HOWL to call this function.

The `onUpDown` argument is either NULL or specifies the name of a `widgetProc` procedure that HOWL will call whenever the user presses an up or down arrow on the control. If this field is NULL, then HOWL will not call a function whenever an up or down arrow is pressed and the application will have to call an appropriate `wUpDown` method to retrieve the current value of the `wUpDown` control.

## 18.2.27 wTimer

```
wTimer
(
    timerID,          // Timer control object name
    period,           // Timeout value in milliseconds
    timing,           // Type of timer (wTimer_t.oneShot or wTimer_t.periodic)
    onTimeOut         // Timeout event handler
);
```

A `wTimer` widget creates a small background thread that calls the `onTimeOut` widget after some period of time. Timers operate on one of two modes: `oneShot` or `periodic`. In the `wTimer_t.oneShot` mode, the timer delays for at least the number of milliseconds specified by the `period` argument and then calls the `onTimeOut` widgetProc exactly once. In the `wTimer_t.periodic` mode, the timer calls the `onTimeOut` widgetProc once every period milliseconds.

Note that declaring a `wTimer` object in the HOWL declarative language does not actually start the timer operating. It initializes the object, but you must call the `wTimer_t.start` method associated with the class to actually begin the timing process. See the description of the `wTimer_t` class later in this document for more details.

## 18.2.28 wWindow..endwWindow

A `wWindow` object is a container. It specifies a rectangular area on a form that contains other objects. All the widgets you declare between a `wWindow` statement and the corresponding `endwWindow` terminator will be contained by the window (and can be treated as a whole) at run time.

```
wWindow
(
    windowID,        // HLA identifier
    caption,         // Window title (ignored unless style calls for title)
    exStyle,         // Extended style for window
    style,           // Windows' style for window
    x,               // x position
    y,               // y position
    w,               // width
    h,               // height
    bkgColor         // RGB background color for window
)

    <<Other widget declarations >>

endwWindow
```



## 18.3 The HOWL Run-time Library

Although the HOWL declarative language (the `wForm...endwForm` macro) makes it very easy to design forms, your applications will need to interact with the HOWL run-time library code in order to make full use of HOWL's capabilities. This section of this document describes the run-time semantics of the HOWL library.

The first thing to note is that HOWL is an object-oriented library. Almost all HOWL data types and code are part of the HOWL object hierarchy. So the best place to start when describing HOWL is with a description of the object hierarchy.

HOWL contains a single base class, appropriately named `wBase_t`. All other objects in HOWL are derived from this class. We'll describe `wBase_t` completely in the next section, but the important thing to note is that `wBase_t` is the root of the class hierarchy tree for HOWL.

As you should know from object-oriented programming, descendant (child/derived) classes inherit all the fields of their base (parent/ancestor) classes. Therefore, all the classes in the HOWL object hierarchy inherit the fields of the `wBase_t` class (and all other ancestor classes to that particular class). In the following sections that describe each of the classes in the HOWL hierarchy, the descriptions will only discuss the fields that are specific to a given class; this document assumes that you understand that each class will inherit fields from all the ancestor classes and that you should look at the documentation for those ancestor classes in order to get the full picture for each class.

Every HOWL object (that is derived from `wBase_t`) contains a special `wType` field. This is an `lword` (128-bit) object that HOWL uses to maintain run-time type information about that particular object. This is an array of 128 bits that specify membership/absence from a particular class. When an application is given a generic pointer to an object of any HOWL type (e.g., `wBase_t`), the application can test this array to see if that object is a specific type (or is derived from a specific type). To accommodate this, HOWL defines a set of constants for each of that HOWL class types (except `wBase_t`) that have the following names and functions:

- `typename_b` ("b" stands for "bit number") is a small integer number between zero and the number of HOWL class types (less than 128) that associates a unique enumerated value with each HOWL class type. This also provides an index into the `wType` bit array.
- `typename_ps` ("ps" stands for "power set") is a singleton set constant containing a "1" bit at index `typename_b` with all other bits containing zero.
- `typename_c` ("c" stands for "constant") is a constant (up to 128 bits) with a "1" bit in each bit position specifying whether `typename` is a descendant (or is) the type indicated by the `typename_b` bit position into `typename_c`. For example, `wProgressBar_c` would contain set bits in bit positions `wProgressBar_b` and `wVisual_p` because `wProgressBar_t` is derived from `wVisual_t`. Because all HOWL objects are derived from `wBase_t`, there is no need to set aside a bit position for `wBase_t` in `wProgressBar_c`. Note that `typename_c` is generated via the logical-OR of `typename_ps` and all the ancestor class "\_ps" values for `typename_t`.

The following diagram shows the HOWL object hierarchy. The nodes in gray are abstract classes; you do not normally create objects of these types (generally, you only create objects of types derived from abstract base classes).



### 18.3.1 Private Data Fields

Many HOWL classes contain private data fields. Although HLA will not prevent you from accessing these private data fields, application programmers should avoid direct access of these private fields. Access to the private data fields is intended for use by HOWL functions only.

For those data fields whose values might be of interest to a HOWL application programmer, the HOWL library generally provides accessor ("getter") and mutator ("setter") functions that let you access these private fields. You should always attempt to use these accessor/mutator functions for all private data access. Reading the values of some private fields (by calling the accessor functions) may cause HOWL to make a Win32 API call to make the value consistent with Windows; writing to a private data field (via a mutator) may cause HOWL to execute some additional code to tell Windows (or the rest of HOWL) about the change. Reading and writing these private data fields directly may circumvent these actions that keep HOWL's internal data structures consistent.

The private fields in a class are easily distinguished in the howl.hhf header file; the header collects all private data fields into record variables within the classes that have a "\_private" suffix. Unless you are writing a class that is an extension of the HOWL library, you should not directly access these fields.

### 18.3.2 Abstract Classes

The HOWL class hierarchy contains several abstract base classes that combine features common to various concrete classes. The following subsections describe each of these base classes.

#### 18.3.2.1 wBase\_t

The wBase\_t class is the root class of the entire HOWL hierarchy. This class has the following definition:

```
wBase_t:
  class

      var
          handle      :dword;
          _name       :string;
          wType       :lword;

      wBase_private:
          record

              visible      :boolean;
              enabled      :boolean;
              onHeap       :boolean;
              align( 4 );

              objectID     :dword;
              nextWidget   :wBase_p;

              // Pointer the wForm object that this
              // object belongs to.

              parentForm   :wForm_p;

              // Handle of the Windows parent window associated
              // with this control. Note that parentForm.handle
              // may not be the same as parentHandle because this
              // object could belong to some other window that
              // is a child window of the main form. (Okay, parentForm
              // was probably a bad name to use).

              parentHandle :dword;
```

```

        endrecord;

static
    objectID_g      :dword;      external( "objectID_object_t" );

// Constructors/Destructors:

procedure create_wBase
(
    wbName :string
); external;

method destroy;                external;
method show;                   external;
method hide;                   external;
method enable;                 external;
method disable;                external;

// Accessor/mutator functions:

method get_handle;             @returns( "eax" );    external;
method get_objectID;          @returns( "eax" );    external;
method get_visible;           @returns( "al" );      external;
method get_enabled;           @returns( "al" );      external;
method get_onHeap;            @returns( "al" );      external;
method get_parentHandle;       @returns( "eax" );    external;
method get_parentForm;        @returns( "eax" );    external;

method set_onHeap( onHeap:boolean );                external;
method set_parentHandle( parentHandle:dword );      external;
method set_parentForm( parentForm:wForm_p );        external;

// Default message processor:

method processMessage
(
    hwnd      :dword;
    uMsg       :dword;
    wParam    :dword;
    lParam    :dword
); external;

endclass;

```

objectID_g	This is a static field that HOWL uses to dynamically assign unique Windows identifiers to objects. Applications should not reference this field; they must not modify the value of this field. HOWL automatically increments this field whenever you create an instance of some HOWL object.
wType	The wType field is a 128-bit bit array that provides run-time type information about an object to the application. If you test bit position <i>typename_b</i> in the wType field, you can determine if the current object is derived from (or is) type <i>typename</i> . The <i>howl.hhf</i> header file defines <i>typename_b</i> constants for all the HOWL types (substituting the appropriate type name, such as <i>wButton_t</i> , for <i>typename</i> ). This is a public field for read-only access. Applications must never modify its value.
handle	Almost all HOWL objects have a Windows handle associated with them. The <i>handle</i> field contains this value. Technically, <i>handle</i> ought to be a private data field (there is even an accessor function for it), however, because applications need to frequently access this



field, it was made public. Note, however, that an application should never write data to this field.

<code>_name</code>	This field is a string representation of the object's name in the main form. This field is mainly useful for testing, debugging, and tracing purposes. Other than initializing this string pointer, the HOWL library doesn't access this field at all, so an application is free to use this field however it wants.
<code>visible</code>	This boolean variable contains true if the object is a <code>wVisual_t</code> object and is visible on the form. It contains false if the object is not visible. If the object isn't a <code>wVisual_t</code> (or descendant) object, then this field's value is meaningless. This is a private field, always use the accessor function to retrieve its value. Applications must never directly change the value of this field.
<code>enabled</code>	This boolean variable contains true if the object is a <code>wVisual_t</code> object and is enabled on the form. It contains false if the object is not enabled. If the object isn't a <code>wVisual_t</code> (or descendant) object, then this field's value is meaningless. This is a private field, always use the accessor function to retrieve its value. Applications must never directly change the value of this field.
<code>onHeap</code>	This field contains true if the object's storage is allocated on the heap. It contains false if the object's storage is not allocated on the heap. If you initialize an object in storage that is not on the heap, it is your responsibility to set this field to false. If you create an object and request heap allocation for it (by calling a class procedure constructor with ESI equal to zero), the HOWL constructors will automatically set this field to true. This is a private field, always use the accessor function to retrieve its value. Applications must never directly change the value of this field.
<code>objectID</code>	The <code>objectID</code> field contains the specific Windows ID (if applicable) for the current object. The create method for an object generally copies the global <code>objectID_g</code> value to this field and then increments the global value to generate unique ID values for each object. For the most part, HOWL ignores this field (it identifies objects by the pointer to the object rather than by the Windows ID). This is a private field, always use the accessor function to retrieve its value. Applications must never directly change the value of this field.
<code>nextWidget</code>	<code>wContainer_t</code> objects use this field to create a linked list of widgets contained by the container object. All objects created via the HOWL declarative language (except the <code>wForm</code> object) are contained by some object (e.g., the <code>wForm</code> object). However, it is possible to dynamically instantiate objects that are not contained by a form, and the form object itself isn't contained by another container, so you cannot assume that this field contains a valid value unless you iterator across the widgets of a container object. This is a private data field, no application program access is legal.
<code>parentForm</code>	This is a pointer to the <code>wForm</code> object that holds the current widget. Note that this is the actual object pointer, not the form's handle. This is a private field, always use the accessor/mutator functions to read/write its value.
<code>parentHandle</code>	This is the window handle of the Windows' object on which the current widget is a child control. This is a private field, always use the accessor/mutator functions to read or write its value.
<code>create_wBase</code>	The <code>create_wBase</code> procedure is the constructor for the class. Because <code>wBase_t</code> is an abstract class, you never instantiate objects of type <code>wBase_t</code> . Unless you are writing a constructor for a new class you've derived from <code>wBase_t</code> , you will probably never call this constructor. This constructor is responsible for setting up the object's <code>_name</code> field (passed as an argument), setting up the <code>ObjectID</code> field, and initializing all the other fields to reasonable default values (that the derived classes' constructors will probably overwrite).
<code>destroy</code>	This is the base level destructor function. You do not generally call this method directly; instead, a higher-level destructor function will probably call this function when you

	invoke <code>destroy</code> on some object. The <code>wBase_t.destroy</code> method checks the <code>onHeap</code> field and will deallocate the storage associated with the object if the storage was allocated on the heap.
<code>show</code> ,	
<code>hide</code>	These two methods simply store true ( <code>show</code> ) or false ( <code>hide</code> ) into the <code>visible</code> field. Other than that, they do nothing. They are included in <code>wBase_t</code> just to allow code to show and hide all objects. Generally, <code>wBase_t</code> descendant classes (usually <code>wVisual_t</code> descendants) override these methods to show or hide a visual object on the screen.
<code>enable</code> ,	
<code>disable</code>	These two methods store true ( <code>enable</code> ) or false ( <code>disable</code> ) into the <code>enabled</code> field.. Other than that, they do nothing. They are included in <code>wBase_t</code> just to allow code to enable and disable objects. Generally, <code>wBase_t</code> descendant classes (usually <code>wVisual_t</code> descendants) override these methods to enable or disable a visual object on the screen.
<code>get_handle</code> ,	
<code>get_objectID</code> ,	
<code>get_visible</code> ,	
<code>get_enabled</code> ,	
<code>get_onHeap</code> ,	
<code>get_parentHandle</code>	
<code>get_parentForm</code>	These are "accessor" functions that retrieve the value of the associated class field. As a general rule you should always call the accessor function to retrieve an object's data field values as the accessor might contain code to "condition" those values prior to consumption
<code>set_onHeap</code>	This is a "mutator" function that lets an application write a value to the <code>onHeap</code> field. Applications should always call this mutator rather than writing directly to the <code>onHeap</code> field because their might be code in the mutator that does additional processing required by the class.
<code>set_parentHandle</code>	This is a "mutator" function that lets an application write a value to the <code>parentHandle</code> field. Applications should be very careful about writing to this field. The only time an application should write to <code>parentHandle</code> is when it dynamically creates a new object at run-time (or, in the rare case of moving a widget from one form to another).
<code>set_parentForm</code>	This is a "mutator" function that lets an application write a value to the <code>parentForm</code> field. Applications should be very careful about writing to this field. The only time an application should write to <code>parentHandle</code> is when it dynamically creates a new object at run-time (or, in the rare case of moving a widget from one form to another).
<code>processMessage</code>	The <code>processMessage</code> method is used by HOWL to do default Windows message processing when no other object handles a message sent from Windows. This is a Windows callback function and you should never call it directly unless you're extending HOWL by added new classes (and you wind up calling this code from the <code>processMessage</code> function in your new class). See the HOWL source code for more details on this function.

### 18.3.2.2 wVisual\_t

The `wVisual_t` class, derived from `wBase_t`, contains the basic information that all visual objects (that is, those appearing on a form) possess. Of course, as `wVisual_t` is derived from `wBase_t`, all `wVisual_t` objects include all the fields from the `wBase_t` type.

```
wVisual_t:
    class inherits( wBase_t );

    var
```

```

align( 4 );
wVisual_private:
    record

        x            :dword;
        y            :dword;
        width         :dword;
        height        :dword;
        bkgColor       :dword;
        bkgBrush       :dword;
        style          :dword;
        exStyle        :dword;

    endrecord;

// Constructors/Destructors:

procedure create_wVisual
(
    wvName           :string;
    parentHandle      :dword;
    x                 :dword;
    y                 :dword;
    width             :dword;
    height            :dword
); external;

// Accessor functions:

method get_x;                @returns( "eax" ); external;
method get_y;                @returns( "eax" ); external;
method get_width;            @returns( "eax" ); external;
method get_height;           @returns( "eax" ); external;
method get_bkgColor;         @returns( "eax" ); external;
method get_style;            @returns( "eax" ); external;
method get_exStyle;          @returns( "eax" ); external;

method set_x( x:dword );      external;
method set_y( y:dword );      external;
method set_width( width:dword ); external;
method set_height( height:dword ); external;
method set_bkgColor
(
    bkgColor:dword
); external;

method move( x:dword; y:dword ); external;
method resize( width:dword; height:dword ); external;

method setFocus;              external;

override method show;         external;
override method hide;         external;
override method enable;       external;
override method disable;      external;
override method destroy;      external;

```

```

        method onClose;
        method onCreate;

        endclass;

x, y, width,
height
    These fields form a bounding rectangle into which the object will appear. These are
    private data fields. HOWL applications should use the accessor/mutator functions to
    change their values.

bkgColor
    This field holds an RGB value representing the background color for the window defined
    by the wVisual_t object. Never access this field directly; always use the accessor/
    mutator functions so that HOWL can properly update the private bkgBrush field. Note all
    objects derived from the wVisual_t class use this field,those objects that do not simply
    ignore its value.

bkgBrush
    This is the brush that Windows uses to paint the background color for the wVisual_t
    object. Note that this is a private field and applications should never access it. HOWL
    computes the value for this field from the bkgColor field. Note all objects derived from
    the wVisual_t class use this field,those objects that do not simply ignore its value.

style
    This is the window style used for various window objects. Not all wVisual_t objects make
    use of this field.

exStyle
    This is the window extended style used for various window objects. Not all wVisual_t
    objects make use of this field.

create_wVisual
    This is the constructor for the class. Because this is an abstract base class, you must never
    call this function with ESI containing NULL. In general, you will never directly call this
    function unless you are creating your own HOWL classes. Whenever you call the
    constructor for a concrete HOWL class, it will automatically call this constructor for you.

get_x,
get_y,
get_width,
get_height,
get_bkgColor
get_style,
get_exStyle

    These are the "accessor" functions for this class. You should call these functions to
    retrieve any data fields for the object.

set_x,
set_y,
set_width,
set_height,
set_bkgColor
set_style
set_exStyle
    These are the "mutator" functions for the class. You must call these functions rather than
    storing values directly into the data fields for the object. These function do additional
    work that is necessary for the system (such as redrawing objects when you change their
    possition or size).

move
    This is a combination of set_x and set_y rolled into a single convenient package (which
    causes less redraw flashing on the screen versus making the two separate calls).

resize
    This is a combination of set_width and set_height rolled into a single convenient package
    (which causes less redraw flashing on the screen versus making the two separate calls).

```

<code>show, hide</code>	These methods make a visual object visible ( <code>show</code> ) or invisible ( <code>hide</code> ) on the form. These methods are also responsible for updating the object's <code>visible</code> field.
<code>enable, disable</code>	These methods enable or disable (gray) an object on the form. Note that not all visual objects can be enabled or disabled. Those that cannot be disabled simply ignore calls to these methods.
<code>set_focus</code>	This method changes the window focus to the current object.
<code>onCreate,</code> <code>onClose</code>	These methods are intended for internal use by the HOWL system. You should never call them directly.

### 18.3.2.3 wClickable\_t

The `wClickable_t` type is an abstract base class derived from `wVisual_t` that contains fields and code associated with objects that the user can click on (with the mouse) on the form. This class inherits all the fields from `wVisual_t`. This class handles both single-click and double-click events. Some objects don't support double-clicking, in which case the double-click facilities wind up being unused.<sup>4</sup>

```
wClickable_t:
  class inherits( wVisual_t );
  var
    align( 4 );
    wClickable_private:
      record

        onClick      :widgetProc;
        onDbClick    :widgetProc;

      endrecord;

  procedure create_wClickable
  (
    wcName      :string;
    parent      :dword;
    x           :dword;
    y           :dword;
    width       :dword;
    height      :dword;
    onClick     :widgetProc
  ); external;

  method get_onClick;      @returns( "eax" );      external;
  method get_onDbClick;   @returns( "eax" );      external;

  method set_onClick( onClick :widgetProc );      external;
  method set_onDbClick( onDbClick :widgetProc );  external;
  method click;          external;

endclass;
```

---

4. Technically, this class should have been split into two classes: `wSingleClickable_t` and `wDoubleClickable_t` (derived from `wSingleClickable_t`) and derived classes that don't support double-clicking would simply be derived from `wSingleClickable_t`. However, it's probably a bit more efficient to implement the single class and ignore double-click operations if they aren't used.

<code>onClick</code>	This is a pointer to a <code>widgetProc</code> procedure that HOWL will call when the user clicks on a <code>wClickable_t</code> object. This field must either contain NULL (meaning HOWL will ignore the click operation) or the address of a <code>widgetProc</code> procedure.
<code>onDbClick</code>	This is a pointer to a <code>widgetProc</code> procedure that HOWL will call when the user double-clicks on a <code>wClickable_t</code> object. If this field contains NULL, then HOWL disables the double-click operation. Note that not all <code>wClickable_t</code> objects support double-clicking, so this field may be ignored by HOWL. Also note that if the user double-clicks on an object (that supports double clicking) and both the <code>onClick</code> and <code>onDbClick</code> fields contain non-NULL values, HOWL will call the <code>onClick widgetProc</code> procedure twice and the <code>onDbClick widgetProc</code> procedure once.
<code>create_wClickable</code>	This is the class constructor. Applications will not normally call this procedure (the constructors for derived classes will call this procedure).
<code>get_onClick,</code> <code>get_onDbClick</code>	These methods return the value of the <code>onClick</code> and <code>onDbClick</code> data fields. Application programs should always call these "accessor" functions rather than directly accessing these fields.
<code>set_onClick,</code> <code>set_onDbClick</code>	These "mutator" functions set the value of the <code>onClick</code> and <code>onDbClick</code> data fields.
<code>click</code>	This method simulates a click on the current object's button.

### 18.3.2.4 `wButton_t`

The `wButton_t` type is an abstract class that contains common fields for all the button, checkbox, and radio button class types. This class is derived from `wClickable_t`, so it inherits all the `wClickable_t` fields.

```

wButton_t:
  class inherits( wClickable_t );

  var
    align( 4 );
    wButton_private:
      record

          onPaint      :widgetProc;
          onHilite      :widgetProc;
          onUnHilite    :widgetProc;
          onDisable     :widgetProc;
          onSetFocus     :widgetProc;
          onKillFocus   :widgetProc;

      endrecord;

  procedure create_wButton
  (
    wbName      :string;
    parent      :dword;
    x           :dword;
    y           :dword;
    width       :dword;
    height      :dword;
    onClick     :widgetProc
  )

```

```

); external;

method get_onPaint;      @returns( "eax" );      external;
method get_onHilite;    @returns( "eax" );      external;
method get_onUnHilite;  @returns( "eax" );      external;
method get_onDisable;   @returns( "eax" );      external;
method get_onSetFocus;  @returns( "eax" );      external;
method get_onKillFocus; @returns( "eax" );      external;

method set_onPaint      ( onPaint      :widgetProc ); external;
method set_onHilite     ( onHilite     :widgetProc ); external;
method set_onUnHilite   ( onUnHilite   :widgetProc ); external;
method set_onDisable    ( onDisable    :widgetProc ); external;
method set_onSetFocus   ( onSetFocus   :widgetProc ); external;
method set_onKillFocus  ( onKillFocus  :widgetProc ); external;

method get_text( txt:string );                  external;
method a_get_text;                                external;
method set_text( txt:string );                  external;

override method processMessage;                  external;

endclass;

```

onPaint	This is a pointer to a <code>widgetProc</code> procedure that HOWL will call whenever the button is painted on the screen. If this pointer contains NULL, HOWL does not call any user-defined <code>onPaint</code> procedure. On older versions of Windows this notification was used for owner-drawn buttons. Newer versions of Windows use the "owner drawn" style for this purpose. By default, the <code>wButton_t.create_wButton</code> constructor initializes this field with NULL.
onHilite	This is a pointer to a <code>widgetProc</code> that HOWL will call (if the pointer is non-NULL) when a button is first pressed. On older versions of Windows, this was used to draw the button in a special depressed state. In HOWL, you can use this event to trigger some operation when the button is first clicked. By default, the <code>wButton_t.create_wButton</code> constructor initializes this field with NULL.
onUnHilite	This is a pointer to a <code>widgetProc</code> that HOWL will call (if the pointer is non-NULL) when a button is released. On older versions of Windows, this was used to draw the button in a normal non-depressed state. In HOWL, you can use this event to trigger some operation when the button is released. By default, the <code>wButton_t.create_wButton</code> constructor initializes this field with NULL.
onDisable	This is a pointer to a <code>widgetProc</code> that HOWL will call (if the pointer is non-NULL) when a button is disabled. On older versions of Windows, this was used to draw the button in a disabled state. In HOWL, you can use this event to trigger some operation when the button is disabled. By default, the <code>wButton_t.create_wButton</code> constructor initializes this field with NULL.
onSetFocus	This is a pointer to a <code>widgetProc</code> that HOWL will call (if the pointer is non-NULL) when focus is shifted to the button. By default, the <code>wButton_t.create_wButton</code> constructor initializes this field with NULL.
onKillFocus	This is a pointer to a <code>widgetProc</code> that HOWL will call (if the pointer is non-NULL) when focus is shifted away from the button. By default, the <code>wButton_t.create_wButton</code> constructor initializes this field with NULL.

Note that `wButton_t` objects inherit the remaining button notification functions, `onClick` and `onDbClick`, from the `wClickable_t` class.

<code>create_wButton</code>	The <code>create_wButton</code> procedure is the constructor for this class. Like all the constructors in HOWL abstract classes, user application do not normally call this constructor directly; constructors in derived classes will call this procedure.
<code>get_onPaint,</code> <code>get_onHilite,</code> <code>get_onUnHilite,</code> <code>get_onDisable,</code> <code>get_onSetFocus,</code> <code>get_onKillFocus</code>	These are accessor functions that return the values of the corresponding <code>widgetProc</code> function pointers. Note that the <code>create_wButton</code> construction initializes all these pointers to NULL when an object is first created.
<code>set_onPaint,</code> <code>set_onHilite,</code> <code>set_onUnHilite,</code> <code>set_onDisable,</code> <code>set_onSetFocus,</code> <code>set_onKillFocus</code>	These are the mutator functions that let you set the addresses of the event handler functions for this class.
<code>get_text</code>	This function retrieves the caption text for a button and stores that text into the string passed as a parameter to this function. The string you pass to this function must have sufficient storage allocated for it to hold the caption or this function will raise an <code>ex.StringOverflow</code> exception.
<code>a_get_text</code>	This function makes a copy of the button's caption on the heap and returns a pointer to this string in the EAX register. It is the caller's responsibility to free the storage associated with this string when it is done using the string data.
<code>set_text</code>	This function changes the button's caption text to the string value you pass as an argument.
<code>processMessage</code>	This is an internal HOWL function. User applications do not call this method.

### 18.3.2.5 wCheckable\_t

The `wCheckable_t` class is an abstract base class for button objects that are "checkable". This includes the various check boxes and radio buttons.

```

wCheckable_t:
    class inherits( wButton_t );

    procedure create_wCheckable_t
    (
        wchkName      :string;
        caption       :string;
        style         :dword;
        parent        :dword;
        x             :dword;
        y             :dword;
        width         :dword;
        height        :dword;
        onClick       :widgetProc
    );    external;

    method set_check( state:dword );                external;
    method get_check; @returns( "eax" );            external;

```



```
endclass;
```

```
create_wCheckable
```

The procedure is the constructor for this class. Like all abstract base class constructors, applications should not call this procedure. The constructors for concrete classes will make calls to this constructor as appropriate.

```
get_check
```

This method retrieves the current state of the checkable button. It returns true in EAX if the button is checked, false if the button is not checked.

```
set_check
```

This method sets the current state of the checkable button. If the single argument contains true, the button will be checked; if the argument contains false, the button will be unchecked.

### 18.3.2.6 wSurface\_t

The `wSurface_t` abstract base class represents a single "window" on a form onto which HOWL can draw things. This abstract class, for example, is the base class for graphic objects like rectangles as well as HOWL views and windows. `wSurface_t` objects are clickable (as this class inherits the `wClickable_t` class).

```
wSurface_t:
  class inherits( wClickable_t );

  var
    align( 4 );
    wSurface_private:
      record

        // onPaint event pointer:

        onPaint      :widgetProc;

      endrecord;

  procedure create_wSurface
  (
    wsName      :string;
    exStyle     :dword;
    style       :dword;
    parent      :dword;
    x           :dword;
    y           :dword;
    width       :dword;
    height      :dword;
    bkgColor    :dword;
    visible     :boolean
  ); external;

  override method destroy;                external;
  override method processMessage;          external;
  override method onClose;                external;
  override method onCreate;               external;

  method get_onPaint;      @returns( "eax" );    external;
  method set_onPaint( onPaint:widgetProc );      external;

endclass;
```

<code>onPaint</code>	This <code>widgetProc</code> pointer is either <code>NULL</code> or points at a procedure that HOWL will call when it receives a <code>w.WM_PAINT</code> message for the surface.
<code>create_wSurface</code>	The <code>create_wSurface</code> procedure is the constructor for the <code>wSurface_t</code> class. Like all abstract base class constructors, applications will not directly call this procedure -- the derived class constructors are the ones that will call this procedure.
<code>destroy</code>	The <code>destroy</code> method is responsible for freeing up the storage associated with the object and the <code>_bkgBrush</code> system resource when an application is done using a <code>wSurface_t</code> object. Normally, applications will not directly call this destructor. Instead, derived class destructor methods will call this method when they are destroyed.
<code>processMessage,</code> <code>onClose,</code> <code>onCreate</code>	These are private methods used by HOWL. Application programs should not call these methods.
<code>get_onPaint,</code> <code>set_onPaint</code>	These are the accessor/mutator functions that get/set the address of the <code>onPaint</code> event-handling <code>widgetProc</code> .

### 18.3.2.7 `wFilledFrame_t`

The `wFilledFrame_t` abstract base class is used for objects that contain graphic entities drawn with a line and filled with an interior color. This includes objects such as rectangles, ellipses, and round rectangles. This extends the `wSurface_t` type by adding a line drawing color and a fill color (on top of the background color provided by `wSurface_t`).

```

wFilledFrame_t:
  class inherits( wSurface_t );
  var
    align( 4 );
    wFilledFrame_private:
      record

        lineColor      :dword;
        fillColor      :dword;

        _linePen       :dword;
        _lineBrush     :dword;
        _fillBrush     :dword;

      endrecord;

  procedure create_wFilledFrame
  (
    wrName      :string;
    parent      :dword;
    x           :dword;
    y           :dword;
    width       :dword;
    height      :dword;
    lineColor   :dword;
    fillColor   :dword;
    bkgColor    :dword
  ); external;

  method get_fillColor;  @returns( "eax" );    external;
  method get_lineColor;  @returns( "eax" );    external;

```

```

method set_fillColor( fillColor:dword );           external;
method set_lineColor( lineColor:dword );           external;

override method destroy;                           external;
override method processMessage;                     external;

endclass;

lineColor      This is the RGB color of the pen used to draw the outline of the graphic object.
                  Applications must not access this field directly but should, instead, use the accessor/
                  mutator functions because those function maintain the private _linePen and
                  _lineBrush fields as well.

fillColor      This is the RGB color of the brush used to fill the interior of the graphic object.
                  Applications must not access this field directly but should, instead, use the accessor/
                  mutator functions because those function maintain the private _fillBrush field as well.
                  Note that the fillColor differs from the background color (inherited from
                  wSurface_t) in that the fill color paints the interior of the graphic object while the
                  background color paints the exterior of the graphic object (within the bounding rectangle).

_linePen,
_lineBrush,
_fillBrush     These are private fields in the class that applications must not access. These fields are
                  automatically maintained by HOWL whenever you call one of the wFilledFrame_t
                  mutator functions.

create_wFilledFrame

                This is the constructor for the wFilledFrame_t class. Like other abstract base classes, you
                do not call this constructor directly, the derived classes' constructions will call this
                constructor for you.

destroy        This is the destructor for the class. It frees up storage allocated for an object and frees up
                the system brush resources created for the object. Applications do not normally call this
                destructor directly; derived class destructors will call this destructor automatically.

get_lineColor
get_fillColor,
set_fillColor,
set_lineColor  These are the accessor and mutator functions for the wFilledFrame_t data fields. You
                must always call these functions to access the data fields of this class because these
                functions also maintain the private pen and brush fields for this class.

```

### 18.3.2.8 wabsEditBox\_t

The `wabsEditBox_t` class is the abstract base class used by the classes that support textual input from the user (e.g., `wEditBox_t`, `wPasswdBox_t`, and `wTextEdit_t`). Edit boxes (and text editors) are among the more feature-rich controls provided by Windows, so it's not surprising that there are many fields and functions associated with this base class.

```

wabsEditBox_t:
    class inherits( wVisual_t );

    var
        align( 4 );
        wabsEditBox_private:
            record

```

```

        onChange      :widgetProc;
        onErrSpace     :widgetProc;
        onHScroll       :widgetProc;
        onVScroll       :widgetProc;
        onMaxText       :widgetProc;
        onUpdate       :widgetProc;
        onSetFocus      :widgetProc;
        onKillFocus     :widgetProc;
        textColor       :dword;

    endrecord;

procedure create_wabsEditBox
(
    webName      :string;
    initialTxt   :string;
    parent       :dword;
    x            :dword;
    y            :dword;
    width        :dword;
    height       :dword;
    style        :dword;
    onChange     :widgetProc
); external;

method get_onChange;      @returns( "eax" );      external;
method get_onErrSpace;   @returns( "eax" );      external;
method get_onHScroll;    @returns( "eax" );      external;
method get_onMaxText;    @returns( "eax" );      external;
method get_onUpdate;     @returns( "eax" );      external;
method get_onSetFocus;   @returns( "eax" );      external;
method get_onKillFocus;  @returns( "eax" );      external;

method set_onChange      ( onChange :widgetProc ); external;
method set_onErrSpace    ( onErrSpace :widgetProc ); external;
method set_onHScroll     ( onHScroll :widgetProc ); external;
method set_onMaxText     ( onMaxText :widgetProc ); external;
method set_onUpdate      ( onUpdate :widgetProc ); external;
method set_onSetFocus    ( onSetFocus :widgetProc ); external;
method set_onKillFocus   ( onKillFocus:widgetProc ); external;

method get_textColor;    @returns( "eax" );      external;

method set_textColor( textColor:dword );        external;

method undo;              external;
method cut;                external;
method copy;              external;
method paste;             external;
method clear;             external;

method get_canUndo;       @returns( "eax" );      external;
method emptyUndoBuffer;   external;

method get_modified;     @returns( "eax" );      external;
method set_modified( modified:boolean );        external;

method get_text( txt:string );                    external;
method a_get_text; @returns( "eax" );            external;
method set_text( txt:string );                    external;

```

```

method get_length; @returns( "eax" );          external;

method get_selectedText( txt:string );          external;
method a_get_selectedText; @returns( "eax" );  external;

method get_selection
(
    var startPosn    :dword;
    var endPosn      :dword
); external;

method set_selection
(
    startPosn    :dword;
    endPosn      :dword
); external;

method replace_selection
(
    replacement :string;
    canUndo     :boolean
); external;

override method processMessage;          external;

endclass;

```

onChange	This is a <code>widgetProc</code> pointer that HOWL will call (if it is not NULL) whenever any change is made to the text associated with the control. Note that HOWL will call this function after the update is drawn to the screen.
onErrSpace	This is a <code>widgetProc</code> pointer that HOWL will call (if it is not NULL) if Windows cannot allocate sufficient storage to handle the current editor operator.
onHScroll	This is a <code>widgetProc</code> pointer that HOWL will call (if it is not NULL) whenever the user clicks on the control's horizontal scroll bar. Note that HOWL will call this function before the update is drawn to the screen.
onVScroll	This is a <code>widgetProc</code> pointer that HOWL will call (if it is not NULL) whenever the user clicks on the control's vertical scroll bar. Note that HOWL will call this function before the update is drawn to the screen.
onMaxText	This is a <code>widgetProc</code> pointer that HOWL will call (if it is not NULL) whenever the user exceeds the maximum number of character for the edit control.
onUpdate	This is a <code>widgetProc</code> pointer that HOWL will call (if it is not NULL) whenever any change is made to the text associated with the control. Note that HOWL will call this function before the update is drawn to the screen (this is the crucial difference between this function and the <code>onChange</code> handler).
onSetFocus	This is a <code>widgetProc</code> pointer that HOWL will call (if it is not NULL) whenever focus shifts to the edit control.
onKillFocus	This is a <code>widgetProc</code> pointer that HOWL will call (if it is not NULL) whenever focus shifts away from the edit control.
textColor	This is the RGB color that Windows will use to draw the text on the editbox. This is a private data field; applications should only access this value using the associated access and mutator. The constructor initializes the text color to black.
get_onChange, set_onChange,	

get_onErrSpace,	
set_onErrSpace,	
get_onHScroll,	
set_onHScroll,	
get_onVScroll,	
set_onVScroll,	
get_onMaxText,	
set_onMaxText,	
get_onUpdate,	
set_onUpdate,	
get_onSetFocus,	
set_onSetFocus,	
get_onKillFocus,	
set_onKillFocus	These are the accessor and mutator functions for all the data fields specific to this abstract base class. Applications should call these functions to access the data fields rather than accessing them directly.
canUndo	This function returns true in EAX if it is possible to undo the last operation to the editBox buffer (via the <code>undo</code> method).
emptyUndoBuffer	This method clears the undo buffer and sets the <code>canUndo</code> flag to false.
undo,	
cut,	
copy,	
paste,	
clear	These methods perform the standard Windows editing functions on the current selection in an edit control. Normally, you'd call these functions when the user selects an appropriate "edit" menu entry or they press one of the standard accelerator keys (e.g., "control-C" for copy).
get_text	This function retrieves the string (text) associated with an editor control and stores the text into the string argument passed as the parameter. The string passed as an argument must be large enough to hold the text or this function will raise an <code>ex.StringOverflow</code> exception.
a_get_text	This function retrieves the string (text) associated with an editor control and stores the text into a string allocated on the heap. It is the caller's responsibility to free the storage associated with this string when it is done using it. This function returns a pointer to the new string in the EAX register.
set_text	This function replaces the text in the edit control with the string passed as an argument.
get_length	This function returns the current number of characters in the string associated with the edit control.
get_selection	This function retrieves the starting and ending zero-based indexes into the string of the current text selection of the edit control. These indexes are returned in the two arguments passed by value. Note that the ending index will contain the offset to the character just beyond the selection in the edit control.
set_selection	This function sets the starting and ending indexes for the edit control.
replace_selection	This function replaces the selected text in the editBox with the string you supply as the argument.
get_textColor,	
set_textColor	These accessor/mutator functions get and set the text color that the widget uses.
processMessage	This is a private method in the <code>wabsEditBox_t</code> class. Applications should not directly call this method.

### 18.3.2.9 wContainer\_t

The `wContainer_t` abstract base class, as its name suggests, is a special class that can contain other widgets. Containers possess a special linked list of `wBase_t` objects and the `wContainer_t` provides methods to manipulate this list of objects. Classes derived from `wContainer_t` include form classes, windows, radio sets, and group boxes.

```

wContainer_t:
  class inherits( wVisual_t );

  var
    align( 4 );
    wContainer_private:
      record

        numWidgets    :uns32;
        widgetList     :wVisual_p;
        lastWidget     :wVisual_p;

      endrecord;

  procedure create_wContainer
  (
    wcName   :string;
    parent   :dword;
    x         :dword;
    y         :dword;
    width     :dword;
    height    :dword
  ); external;

  override method destroy;          external;
  override method show;             external;
  override method hide;             external;
  override method enable;           external;
  override method disable;          external;

  method get_numWidgets; @returns( "eax" ); external;
  method insertWidget( theWidget:wBase_p ); external;
  method findWidget( objectID:dword ); external;
  iterator widgetOnForm( nestingLevel:uns32 ); external;
  iterator widgetsJustOnForm;       external;

endclass;

```

<code>numWidgets</code>	This is the number of widgets contained by the <code>wContainer_t</code> object. The constructor initializes this field to zero and inserting widgets into the container increments this field by one. This is a private field; use the <code>get_numWidgets</code> method to retrieve this field's value. Applications should never store a value directly into this field.
<code>widgetList</code>	This is a pointer to the first item in the list of widgets contained by the <code>wContainer_t</code> object. This is a private field; applications should never access this field.
<code>lastWidget</code>	This is a pointer to the last item in the list of widgets contained by the <code>wContainer_t</code> object. This is a private field; applications should never access this field.

<code>destroy</code>	This method iteratively calls the destructor for all widgets contained by the <code>wContainer_t</code> object and then it frees the storage held by the container object itself. Because <code>wContainer_t</code> is an abstract base class, applications should not call this destructor directly. Instead, they will call the destructor for some derived class which will indirectly call this method. Important note: because a container automatically calls the destructor for all widgets contained by the container, an application must not explicitly call the destructor for any of those widgets.
<code>show</code>	This method iteratively calls all the <code>show</code> methods for each of the widgets contained by the container. Because <code>wContainer_t</code> is an abstract base class, applications should not call this method directly. Instead, they will call the <code>show</code> method for some derived class which will indirectly call this method.
<code>hide</code>	This method iteratively calls all the <code>hide</code> methods for each of the widgets contained by the container. Because <code>wContainer_t</code> is an abstract base class, applications should not call this method directly. Instead, they will call the <code>hide</code> method for some derived class which will indirectly call this method.
<code>enable</code>	This method iteratively calls all the <code>enable</code> methods for each of the widgets contained by the container. Because <code>wContainer_t</code> is an abstract base class, applications should not call this method directly. Instead, they will call the <code>enable</code> method for some derived class which will indirectly call this method.
<code>disable</code>	This method iteratively calls all the <code>disable</code> methods for each of the widgets contained by the container. Because <code>wContainer_t</code> is an abstract base class, applications should not call this method directly. Instead, they will call the <code>disable</code> method for some derived class which will indirectly call this method.
<code>get_numWidgets</code>	This method returns the value of the <code>numWidgets</code> field in the EAX register. Note that there is no corresponding "set_numWidgets" mutator function; applications cannot directly set the value of this field, <code>wContainer_t</code> objects increment the value of this field by calling the <code>insertWidget</code> method.
<code>insertWidget</code>	This method inserts a widget into the <code>wContainer_t</code> 's linked list. The argument is a pointer to a <code>wVisual_t</code> object (or some object type derived from <code>wVisual_t</code> ). Widgets are inserted into the linked list at the end of the list (i.e., after the widget pointed at by <code>lastWidget</code> ). As this is being written, there is no way to remove a widget from a <code>wContainer_t</code> 's widget list. This restriction may be relaxed in a future version of HOWL.
<code>findWidget</code>	This function searches for a widget in the <code>wContainer_t</code> 's widget list. The single argument is the <code>ObjectID</code> value (inherited from <code>wBase_t</code> ) of the object to search for. This function is mainly useful for various Windows callback functions (message handlers) that pass along a widgets object identifier without specifying the object itself.
<code>widgetOnForm</code>	This is an HLA iterator (that you use in an HLA <b>foreach</b> loop) that iterates over all the widgets in a container's widget list. This iterator is recursive. This means that if one of the items in a widget list is a <code>wContainer_t</code> class (or a class derived from <code>wContainer_t</code> ), then the iterator will drill down into that container and return its list of widgets as well. For example, the <code>wContainer_t</code> <code>destroy</code> , <code>show</code> , <code>hide</code> , <code>enable</code> , and <code>disable</code> methods all use this iterator to process all the widgets held by the container. On each iteration of the foreach loop, this iterator returns a pointer to the current widget in the EAX register.
<code>widgetsJustOnForm</code>	This iterator is very similar to <code>widgetOnForm</code> except that it is not recursive. On each iteration of the foreach loop it will return an entry from the current container's widget list. It will not recursively process the lists of any <code>wContainer_t</code> objects appearing in the current container.



## 18.3.3 Containers

There are five main (concrete) container objects in HOWL: `wForm_t` objects, `window_t` objects, `wGroupBox_t` objects, `wTabs_t` objects, and `wRadioSet_t` objects. We'll consider the first three of these objects in this section (plus menu objects, because it makes sense to discuss menus along with `wForm_t` objects) and `wRadioSet_t` objects in the section on buttons.

### 18.3.3.1 Forms and Menus

The main form for an application is a `wForm_t` object. Most applications will have a single `wForm_t` object, though a multi-window application can certainly support two or more `wForm_t` windows. A `wForm_t` menu is special (compared, say, to a `wTabPage_t` object) because it supports a menu. In a sense, a `wForm_t` object is a double container because it can contain an arbitrary list of widgets and it can contain a list of menu items.

#### 18.3.3.1.1 `wForm_t`

The `wForm_t` class type is a `window_t` object with the addition of a list of menu items (which may be empty). The `wForm` statement in the HOWL declarative language defines a class that is derived from `wForm_t`. The `wForm` statement inserts the widget declarations into this new class and creates a constructor for the new class. Therefore, `wForm_t` is the basis for all forms created with the HOWL declarative language. Technically, `wForm_t` is a concrete class, not an abstract class, (meaning you can create objects of type `wForm_t`). However, in most HOWL applications your main window will actually consist of an object whose type is derived from `wForm_t` (the `wForm..endwForm` declaration creates this class for you).

```
wForm_t:
  class inherits( window_t );
  var
    align( 4 );
    wForm_private:
      record

        menuList      :wMenuItem_p;

      endrecord;

  procedure create_wForm
  (
    wwName      :string;
    caption     :string;
    exStyle     :dword;
    style       :dword;
    parent      :dword;
    x           :dword;
    y           :dword;
    width       :dword;
    height      :dword;
    fillColor   :dword;
    visible     :boolean
  ); external;

  method appendMenuItem( mi:wMenuItem_p );    external;
  override method insertWidget;               external;
  override method processMessage;             external;

endclass;
```

`menuList`            This is a private data field that points at the list of menu items for the `wForm_t`'s menu. Applications should not access this field.

`create_wForm`

This is the constructor for `wForm_t` objects. If you call this procedure using the classname, e.g., "`wForm_t.create_wForm( ... );`" then this constructor will allocate storage for the `wForm_t` object on the heap and initialize all the fields of the `wForm_t` object with reasonable values (including the parameter values you specify). If you have a statically declared `wForm_t` object, or a pointer to a `wForm_t` object you've already allocated storage for, then calling this procedure via that object will initialize the fields of that object, e.g., "`somewFormObject.create_wForm( ... );`", without allocating new storage for the object. Although `wForm_t` is a concrete class and it's not unreasonable for an application to call `create_wForm` directly, most applications will actually work with classes derived from `wForm_t`, so it would be unreasonable for an application to call this constructor directly. If you look back at the discussion of the `wForm.endwForm` statement in the section on the HOWL declarative language, you'll notice that the `appStart` procedure calls a constructor named "`myForm.create_myForm`". This constructor is a good example of a constructor for a class (`myForm_t`) derived from `wForm_t`. Note that `myForm` is a statically declared object of type `myForm_t` (which is derived from `wForm_t`) in the example given earlier in this documentation.

**wwName:** this is a string that HOWL stores in the `_name` field of the object (from `wBase_t`). HOWL does not copy this string, so the character data associated with this argument must exist for the duration of the program (and must not change). You should use `str.a_cpy` to create a copy of this string to pass to `create_wForm` if the original string might change during the execution of the program.

**caption:** This is the string that HOWL will display in the title bar of the `wForm_t`'s window on the screen. Note that Windows will create an internal copy of this string, so it need not continue to exist after the call to the constructor.

**exStyle:** the constructor logically-ORs the value you supply to this parameter with the Windows' `w.WS_EX_CONTROLPARENT` extended window style. Normally you would supply zero for this parameter value. However, if you want your form to have some additional window extended style attributes, you can supply one (or more) of the `w.WS_EX_*` constants here. See the Windows documentation for `w.CreateWindowEx` for more details on the possible extended style constants you can use.

**style:** the constructor logically-ORs this value with the (`w.WS_CLIPCHILDREN` | `w.WS_OVERLAPPEDWINDOW`) style when creating the window. Normally you would supply zero for this parameter value. However, if you want your form to have some additional window style attributes, you can supply one (or more) of the `w.WS_*` constants here. See the Windows documentation for `w.CreateWindow` for more details on the possible window style constants you can use.

**parent:** this is the handle of the parent window for this form. This should always be `NULL` (if you are creating a child window, you'll probably be using the `window_t` type, not a `wForm_t` type).

**x, y, width, height:** These fields describe the position and size of the `wForm_t` window on the main screen. These will either be the pixel coordinates and sizes or the Windows' constant `w.CW_USEDEFAULT` (that tells Windows to pick good default values for these arguments).

**fillColor:** This is the RGB background color you want to use for the client (drawing) area of the `wForm_t` window.

**visible:** if true, then the constructor makes this form visible when it creates it. If this argument is false, then you must explicitly call the `show` method to make the form visible on the screen. For the main form, this argument is almost always true. If you are creating multiple windows (forms), then you might set this argument to false for all but the main form and call the `show` method to display the windows as needed.

`appendMenuItem`

This method appends a new menu item to the `wForm_t` object's menu list. See the discussion of menu items (following shortly) for a complete discussion of those objects. In most HOWL applications, you will not directly call this method; the HOWL declarative language automatically appends all menu items you declare in the `wMenu.endwMenu`

	statement to the main window's <code>wForm_t</code> object. However, if you want to manually create a <code>wForm_t</code> object, you can call this function to attach a menu item to the form.
<code>insertWidget</code>	See <code>wContainer_t.insertWidget</code> for more details. Note that this overridden version will also set the <code>parentForm</code> field of the widget you insert (plus all contained widgets, if the argument is a container) to the value of the form.
<code>processMessage</code>	This is an internal HOWL method. Applications should not call this function.

### 18.3.3.1.2 `wMenu_t`

Exactly one `wMenu_t` object is create for every `wForm_t` object that has a menu. The `wMenu_t` object corresonds to the main menu for the form. This should be the first menu item (note that `wMenu_t` is derived from `wMenuItem_t`, described next) added to the `wForm_t` object via an `appendMenuItem` method call.

```
wMenu_t:
  class inherits( wMenuItem_t );

  // Constructors/Destructors:

  procedure create_wMenu
  (
    wmName           :string;
    wmText           :string;
    parentHandle     :dword
  ); external;

  override method destroy;          external;

endclass;
```

<code>create_wMenu</code>	<p>This constructor creates the main menu item.</p> <p><b>wmName:</b> this is a string that HOWL stores in the <code>_name</code> field of the main menu object (from <code>wBase_t</code>). HOWL does not copy this string, so the character data associated with this argument must exist for the duration of the program (and must not change). You should use <code>str.a_cpy</code> to create a copy of this string to pass to <code>create_wMenu</code> if the original string might change during the execution of the program.</p> <p><b>wmText:</b> This is basically ignored and should be the empty string or some string like "main menu".</p> <p><b>parentHandle:</b> This must be the handle of the <code>wForm_t</code> object that contains this menu.</p>
<code>destroy</code>	<p>This is the destructor method for the <code>wMenu_t</code> object. Applications should not call this method directly if the menu is on a form. The <code>wForm_t</code> object that holds the menu will automatically call the destructors for all the objects on its menu list (including the <code>wMenu_t</code> object).</p>

### 18.3.3.1.3 `wMenuItem_t`

`wMenuItem_t` objects generally correspond to the actual menu items present in the main window.

```
wMenuItem_t:
  class inherits( wBase_t );

  var
    align( 4 );
    wMenuItem_private:
      record
```

```

        nextMenu      :wMenuItem_p;
        itemType      :dword;
        itemString     :string;
        itemHandler    :widgetProc;

    endrecord;

// Constructors/Destructors:

procedure create_wMenuItem
(
    wmiName      :string;
    parentHandle :dword;
    itemType     :dword;
    itemString   :string;
    itemHandler  :widgetProc
); external;

override method enable;                external;
override method disable;               external;

method checked( state:boolean );        external;

// Accessor functions:

method get_itemType;      @returns( "eax" );    external;
method get_itemString;    @returns( "eax" );    external;
method get_itemHandler;   @returns( "eax" );    external;

method set_itemType( itemType:dword );        external;
method set_itemString( itemString:string );    external;
method set_itemHandler( itemHandler:widgetProc ); external;

endclass;

nextMenu      This is a private data field that the wForm_t class uses to create a linked list of menu items
               on the main form. Applications should not access this field.

parentHandle  This is the handle of the main application's form

itemType      This field specifies the type of the menu item. It must be the logical OR of one or more of
               the following constants: w.MF_STRING, w.MF_CHECKED, w.MF_DISABLED,
               w.MF_ENABLED, w.MF_GRAYED, w.MF_SEPARATOR, and w.MF_UNCHECKED. See the
               Windows documentation for w.AppendMenu for more details.

itemString    This is the string text that Windows displays for the menu item.

itemHandler   This is a widgetProc procedure that HOWL will call when the user selects a menu item.

enable,
disable       These two methods will enable or disable a menu item, respectively. Note that when
               HOWL disables a menu item, it will gray that menu item and will prevent the user from
               selecting it in the menu. This is equivalent to the (w.MF_DISABLE | w.MF_GRAYED)
               item type.

```

checked	If the current menu item has the <code>w.MF_CHECKED</code> flag, then calling this method will display a check mark if the argument is true, it will clear a displayed check mark if the argument is false.
get_itemType,	
get_itemString,	
get_itemHandler	These accessor functions return the values of the respective fields in the EAX register.
set_itemType,	
set_itemString,	
set_itemHandler	These mutator functions set the values of the respective fields to the value passed as an argument. Note that the <code>set_itemString</code> function does not make a copy of the string data, it stores the string pointer directly into the <code>itemString</code> data field. Therefore, your application should make a copy of the string to pass to this mutator if the string data could change.

### 18.3.3.2 Tabbed Forms

A tabbed form is a `wForm_t` object that contains exactly one `wTabs_t` object on it. A `wTabs_t` object is a container that contains a list of `wTabPage_t` (window) objects on it, one `wTabPage_t` object for each tab present on the `wTabs_t` control. Whenever the user selects one of the tabs on the `wTabs_t` control, HOWL will display the corresponding `wTabPage_t` object (hiding the previously displayed `wTabPage_t` object). This lets an application have multiple pages on the main form that the user can select the pages as needed.

#### 18.3.3.2.1 wTabs\_t

The `wTabs_t` class is a special type of container class. If used correctly, `wTabs_t` objects only hold `wTabPage_t` objects (called "pages") that correspond to a window on top of the main form that hold the widgets associated with a tab on that main form. `wTabs_t` objects represent the current state of a tabbed form.

```

wTabPage_array :pointer to wTabPage_p;

wTabs_t:
  class inherits( wContainer_t );

  var
    align( 4 );
    wTabs_private:
      record

        curSelection      :dword;
        numTabs           :uns32;
        pages              :wTabPage_array;
        numElements       :dword;

      endrecord;

  procedure create_wTabs
  (
    wtName      :string;
    parent      :window_p;
    x           :dword;
    y           :dword;
    width       :dword;
    height      :dword
  ); external;

  method get_numTabs;                @returns( "eax" ); external;
  method curTab;                     @returns( "eax" ); external;

```

```

method setTab( tab:uns32 );                                external;
method get_page( tabIndex:dword ); @returns( "eax" );      external;
method deleteTab( tabIndex:dword ); @returns( "eax" );      external;
method insertTab
(
    index    :dword;
    tabText  :string;
    page     :wTabPage_p
); external;

    override method destroy;                                external;
    override method processMessage;                          external;

endclass;

```

curSelection	This field contains the zero-based tab index for the currently active tab on a <code>wForm_t</code> tabbed object. This is a private data field that applications must not access.
numTabs	This private data field contains the number of tabs currently associated with the <code>wTabs_t</code> object. Applications should never directly access this data field. They can use the <code>get_numTabs</code> accessor method to query its value. Applications must never directly change the value of this field.
pages	The <code>pages</code> data field is a pointer to an array of <code>wTabPage_p</code> pointers. This is a private data field and applications should not access or modify its contents. Applications can use the <code>get_page</code> method to read entries from the array pointed at by pages.
numElements	This is a private data field that specifies the number of elements in the array pointed at by the <code>pages</code> data field. Applications must not access or modify this value.
create_wTabs	<p>This is the constructor for the <code>wTabs_t</code> class. If called as a class procedure (e.g., "<code>wTabs_t.create_wTabs</code>") then this procedure will allocate storage on the heap for a new <code>wTabs_t</code> object and return a pointer to that new object in ESI. If you call this method via an object variable (e.g., "<code>myTab.create_wTab</code>") then this constructor will initialize the fields of that object without allocating new storage for it (and return a pointer to the object in ESI).</p> <p><b>wtName:</b> this string is assigned to the <code>_name</code> field of the object. This string should not change during the execution of the program. Pass a copy of the string (using <code>str.a_cpy</code>) if it is possible for this string to change during program execution.</p> <p><b>parent:</b> this is the handle of the parent <code>window_t</code> object (e.g., <code>wForm_t</code> object) that holds this <code>wTabs_t</code> object.</p> <p><b>x, y, width, height:</b> These arguments specify the position and size of the tab control within the client area of the parent object. The x and y values are almost always zero, the width should be the width of the client area of the parent window, and the height should 25 or some other similar value.</p>
destroy	This is the destructor for the <code>wTabs_t</code> class. Generally, tabs are attached to a <code>wForm_t</code> object and that object will destroy the tabs when it is destroyed. Therefore, an application will rarely call a <code>wTabs_t</code> destructor unless it explicitly creates the <code>wTab_t</code> object and doesn't attach those tabs to some other container (e.g., <code>wForm_t</code> object).
get_numTabs	This method returns the current number of tabs on the tab control. In addition to returning the value of the <code>numTabs</code> data field, this method also does a sanity check to ensure that the number of tabs on the Windows control matches the <code>numTabs</code> data field value.
curTab	This method returns the currently selected tab index (0.. <code>numtabs</code> -1). Note that this method actually calls Windows to retrieve this value, it is not simply an accessor to the <code>curSelection</code> data field (indeed, this is a mutator to the <code>curSelection</code> field because it will update <code>curSelection</code> with the value that Windows returns). This also does a sanity check on the values and raises an exception if Windows and HOWL have different ideas about the number of tabs on the control.

setTab	This method sets the currently selected tab to the value you pass as an argument. This value must be in the range 0..numTabs-1. If the argument is outside this range, the setTab method raises an exception. This method also does a sanity check to ensure that Windows' and HOWL's tab counts are the same.
deleteTab	This function deletes a tab (specified by the zero-based <code>tabIndex</code> argument) from the tab bar on the form. Note that this function does not call the destructor for the tab object, nor does it destroy any of the widgets contained on the form. It simply removes the tab from the tab bar (and the corresponding entry from the array pointed at by the <code>pages</code> data field). This function returns the pointer to the <code>wTabPage_t</code> widget removed from the <code>pages</code> array in the EAX register. Note that this method does not remove the <code>wTabPage_t</code> object from the tab's <code>widgetList</code> (that is, the tab still contains the <code>wTabPage_t</code> object). Therefore, if you destroy the tab, or otherwise iterate over all the widgets held by the <code>wTabs_t</code> container, you will still process the deleted tab. All that deleting a tab does is visibly remove it from the tab control on the form. Note that you can insert the deleted <code>wTabPage_t</code> object (whose address is returned in EAX by <code>deleteTab</code> ) by calling <code>insertTab</code> . It is your responsibility to save the value returned by <code>deleteTab</code> (or otherwise locate the deleted item) if you intend to reinsert it into the tab control later on.
insertTab	<p>This build adds a new tab entry to the tab control and inserts a pointer to a <code>wTabPage_t</code> object into the array pointed at by the tab's <code>pages</code> data field.</p> <p><b>index:</b> This is the zero-based index specifying the tab position. This value must be in the range 0..numTabs. If <code>index</code> is less than <code>numTabs</code>, then <code>insertTab</code> will insert the new tab in front of the tab at the specified index. If <code>index</code> is equal to <code>numTabs</code>, then <code>insertTab</code> will append the new tab to the end of the tab list. If <code>index</code> is greater than <code>numTabs</code>, <code>insertTab</code> will raise an exception.</p> <p><b>tabText:</b> this is the string that Windows will draw on the tab. Windows will make a copy of this string's character data.</p> <p><b>page:</b> this is the <code>wTabPage_t</code> object that HOWL will display when you select the new tab. Generally, a program will place several other widgets on this <code>wTabPage_t</code> display surface</p>
processMessage	This is a private method. Applications should never call this method..

### 18.3.3.3 wGroupBox\_t

A `wGroupBox_t` object is a rectangular panel with a caption along the upper-left-hand corner of the rectangle (on top of the line outlining the rectangle). Generally, `wGroupBox_t` objects are used to visually separate and group items on a form.

```

wGroupBox_t:
    class inherits( wContainer_t );

    procedure create_wGroupBox
    (
        wgbName      :string;
        caption      :string;
        parent       :dword;
        x            :dword;
        y            :dword;
        width        :dword;
        height       :dword
    ); external;

endclass;

```

`create_wGroupBox` This is the constructor for the `wGroupBox_t` class.

**wgbName:** This is the name of the object that HOWL stores into the `_name` data field.

**caption:** This is the string that Windows displays in the upper-left-hand corner of the group box. Windows makes a copy of this string for its internal use.

**parent:** This is the handle of the window (usually the main form or a `wView_t` object on a tabbed form) that contains the group box.

**x, y, width, height:** These are the (parent-form-relative) coordinates and size for the group box.

## 18.3.4 Graphic Objects

Graphic objects in HOWL are static images that HOWL draws on a form. Examples include rectangles and ellipses. All graphic objects are derived from `wSurface_t`, which is derived from `wClickable_t`, so graphic objects can respond to single clicks. Note that although the `wClickable_t` type also handles double clicks, graphic objects don't send double-click notifications, so if you try to install a double-click handler for one of these objects (which is legal to do), it won't have any effect. Double-clicks will simply be treated as two single clicks. Because attaching an "onClick" handler to a graphic object is not the common case, you will have to explicitly call the `set_onClick` method to initialize an `onClick` handler. The HOWL declarative language doesn't provide an option to do this for you.

### 18.3.4.1 wBitmap\_t

The `wBitmap_t` class lets you create objects that display a bit mapped image on a form.

```
wBitmap_t:
  class inherits( wSurface_t );

  var
    align( 4 );
    wBitmap_private:
      record

        stretch          :boolean;
        align( 4 );

        imageName         :string;
        imageHandle       :dword;
        sourceX           :dword;
        sourceY           :dword;
        sourceW           :dword;
        sourceH           :dword;
        destW             :dword;
        destH             :dword;

      endrecord;

  procedure create_wBitmap
  (
    wiName      :string;
    imageName   :string;
    parent      :dword;
    x           :dword;
    y           :dword;
    width       :dword;
    height      :dword;
    bkgColor    :dword
  ); external;

  method get_imageName;      @returns( "eax" );      external;
```



```

method get_sourceX;           @returns( "eax" );      external;
method get_sourceY;           @returns( "eax" );      external;
method get_sourceW;           @returns( "eax" );      external;
method get_sourceH;           @returns( "eax" );      external;
method get_destW;             @returns( "eax" );      external;
method get_destH;             @returns( "eax" );      external;
method get_stretch;           @returns( "al" );       external;

method load_bitmap( imageName:string );              external;
override method destroy;                             external;
override method processMessage;                      external;

method normalBitmap;           @returns( "eax" );      external;
method stretchBitmap
(
    sourceX      :dword;
    sourceY      :dword;
    sourceW      :dword;
    sourceH      :dword;
    destW        :dword;
    destH        :dword
); external;

```

endclass;

**imageName** This is a string that specifies the name of the bitmap resource within the executable file. Important this is not the name of a ".bmp" file on the disk. You must compile ".bmp" files into your executable using a "resource compiler". The name you attach to the resource is the name that this string will contain. This is a private field. Applications should not access it directly. Note that if the value of this field is less than \$1\_0000, then it specifies a standard Windows bitmap resource rather than an actual resource name. See the discussion of the `load_bitmap` method for more details.

**imageHandle,**

**sourceX,**

**sourceY,**

**sourceW,**

**sourceH,**

**destW,**

**destH**

These are private data fields that applications must not access or modify.

**stretch**

This private field determines whether a bitmap is stretched or displayed normal. This field is set to true by the `stretchBitmap` method and set to false by the `normalBitmap` method.

**create\_wBitmap**

This is the constructor for the `wBitmap_t` class. If called as a class procedure (e.g., "`wBitMap_t.create_wBitmap`") this procedure will allocate storage on the heap for the object and return a pointer to the new (initialized) object in ESI. If you call this constructor specifying an existing object, then it will simply initialize that object in-place.

**wiName:** this is the string that the HOWL code stores into the `_name` field. This string's value should not change over the lifetime of the bitmap object.

**imageName:** This is either a string containing a bitmap resource name within the executable file, or a standard Windows bitmap resource value. Legal bitmap resource constants are:

`w.OBM_BTNCORNERS`, `OBM_BTSIZE`, `w.OBM_CHECK`, `w.OBM_CHECKBOXES`,  
`w.OBM_CLOSE`, `w.OBM_REDUCE`, `w.OBM_COMBO`, `w.OBM_REDUCED`,  
`w.OBM_DNARROW`, `w.OBM_RESTORE`, `w.OBM_DNARROWD`,

w.OBM\_RESTORED, w.OBM\_DNARROWI, w.OBM\_RGARROW,  
w.OBM\_LFARROW, w.OBM\_RGARROWD, w.OBM\_LFARROWD,  
w.OBM\_RGARROWI, w.OBM\_LFARROWI, w.OBM\_SIZE, w.OBM\_MNARROW,  
w.OBM\_UPARROW, w.OBM\_UPARROWD, w.OBM\_UPARROWI, w.OBM\_ZOOM,  
w.OBM\_ZOOMD.

See the Windows documentation for more details on these constants.

**parent:** this is the handle of the form, `wView_t`, or other drawing surface that contains the `wBitmap_t` object (and on whose surface the bitmap will be drawn).

**x, y, width, height:** these arguments specify the bounding box on the parent's form where the bitmap will be drawn. If this bounding rectangle is larger than the bitmap image (in any dimension), then HOWL will fill the unaccounted-for area with the background color. If this bounding rectangle is smaller than the image (in any dimension), then HOWL will clip the bitmap when drawing it.

**bkgColor:** this is the RGB background color that HOWL uses to fill in the bounding rectangle if the bitmap is smaller than the bounding rectangle. If the H.O. byte of this color value contains \$FF, then HOWL will use the RGB value in the L.O. three bytes to fill the initial object window's background color but will not redraw the background thereafter (making the object's background transparent).

<code>load_bitmap</code>	This method loads the bitmap object with an image resource in the executable file. The argument is a string specifying the resource name or one of the standard Windows bitmap resource constants (see the discussion in <code>create_wBitmap</code> ). Note that this is not a ".bmp" filename. You must compile bitmaps into the executable file using a resource compiler and specify the resource name as the parameter to <code>load_bitmap</code> .
<code>destroy</code>	This is the class destructor. This method releases all resources and memory in use by the <code>wBitmap_t</code> object. Usually, applications will not call this method directly. Instead, HOWL will automatically call this destructor when destroying the main application's <code>wForm_t</code> form or a <code>wView_t</code> object that contains the bitmap.
<code>processMessage</code>	This is a private method that applications should never call.
<code>normalBitmap</code>	This method sets the <code>stretch</code> field to false to display the bitmap in a normal form.
<code>stretchBitmap</code>	This method sets the <code>stretch</code> field to true and copies the parameters to the corresponding private data fields.  <b>sourceX:</b> The stretched bitmap will be copied from the original bit map starting at this zero-based x-coordinate.  <b>sourceY:</b> The stretched bitmap will be copied from the original bit map starting at this zero-based y-coordinate.  <b>sourceW:</b> This many bits along the X axis will be copied to the stretched bitmap.  <b>sourceH:</b> This many bits along the Y axis will be copied to the stretched bitmap.  <b>destW:</b> The bit mapped will be stretched (or shrunk) so that the <code>sourceW</code> bits will be displayed using <code>destW</code> bits.  <b>destH:</b> The bit mapped will be stretched (or shrunk) so that the <code>sourceH</code> bits will be displayed using <code>destH</code> bits.

### 18.3.4.2 wEllipse\_t

Ellipse graphic objects (of which circles are special cases) allow you to place ellipses anywhere on a form or `wView_t` object.

```
wEllipse_t:
    class inherits( wFilledFrame_t );

    procedure create_wEllipse
    (
```

```

        wrName      :string;
        parent      :dword;
        x           :dword;
        y           :dword;
        width       :dword;
        height      :dword;
        lineColor    :dword;
        fillColor    :dword;
        bkgColor     :dword
    );  external;

    override method processMessage;                external;

endclass;

```

**create\_wEllipse** This is the constructor for the `wEllipse_t` class. If you call this as a class procedure (e.g., "`wEllipse_t.create_wEllipse`") then this procedure will allocate storage for a new `wEllipse_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wEllipse` will initialize that object in-place.

**wrName:** HOWL assigns this string to the `_name` data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

**parent:** this is the handle of the `wView_t` or `wForm_t` object on which the ellipse will be drawn.

**x, y, width, height:** These arguments form a bounding box in which the ellipse will be drawn. If `width` and `height` are the same value (meaning the bounding rectangle is a square), then the ellipse will form a circle.

**lineColor:** this is the RGB color value for the pen that HOWL will use to draw the outline of the ellipse.

**fillColor:** this is the RGB color value for the brush that HOWL will use to paint the interior of the ellipse.

**bkgColor:** this is the RGB color value for the brush that HOWL will use to paint the exterior of the ellipse. If the H.O. byte of this color value contains \$FF, then HOWL will use the RGB value in the L.O. three bytes to fill the initial object window's background color but will not redraw the background thereafter (making the object's background transparent).

**processMessage** This is a private method that applications must not call.

### 18.3.4.3 wPie\_t

The `wPie_t` graphic object draws a slice of a pie graph on a window. Note that `wPie_t` is only capable of drawing a single wedge of a pie graph. **Note:** the procedures and methods in the `wPie_t` class make use of the FPU on the CPU. You must ensure that the FPU is initialized (i.e., you're not in MMX mode) before using these functions.

```

wPie_t:
    class inherits( wFilledFrame_t );

    var
        align( 8 );
        wPie_private:
            record

                startAngle    :real64;
                endAngle       :real64;

```

```

        endrecord;

procedure create_wPie
(
    wrName      :string;
    parent      :dword;
    x           :dword;
    y           :dword;
    width       :dword;
    height      :dword;
    startAngle   :real64;
    endAngle    :real64;
    lineColor    :dword;
    fillColor    :dword;
    bkgColor     :dword
); external;

override method processMessage;                external;

method get_startAngle; @returns( "st0" );        external;
method get_endAngle;   @returns( "st0" );        external;

method set_startAngle( startAngle:real64 );      external;
method set_endAngle( endAngle:real64 );          external;

endclass;

```

**startAngle** This is the starting angle (measure counter-clockwise from the vertical line) from which wPie\_t objects between drawing the wedge. Applications should not access or modify this field directly; they should use the supplied accessor and mutator functions for this purpose. Do not assume the value of this field is degrees or radians.

**endAngle** This is the ending angle (measured counter-clockwise from the vertical line) to which wPie\_t object draw the wedge (from the startAngle to the endAngle in the counter-clockwise direction). Applications should not access or modify this field directly; they should use the supplied accessor and mutator functions for this purpose. Do not assume the value of this field is degrees or radians.

**create\_wPie** This the is the constructor for the wPie\_t class. If you call this as a class procedure (e.g., "wPie\_t.create\_wPie") then this procedure will allocate storage for a new wPie\_t object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then create\_wPie will initialize that object in-place.

**wrName:** HOWL assigns this string to the \_name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

**parent:** this is the handle of the wView\_t or wForm\_t object on which the wedge will be drawn.

**x, y, width, height:** These arguments form a bounding box in which the pie wedge will be drawn.

**startAngle:** starting angle for the wedge (see the discussion above). This angle is specified in degrees (not radians).

**endAngle:** ending angle for the wedge (see the discussion above). This angle is specified in degrees, not radians.

**lineColor:** this is the RGB color value for the pen that HOWL will use to draw the outline of the wedge.

**fillColor:** this is the RGB color value for the brush that HOWL will use to paint the interior of the wedge.

**bkgColor:** this is the RGB color value for the brush that HOWL will use to paint the exterior of the wedge.

`processMessage` This is a private method that applications must not call.

`get_startAngle,`

`get_endAngle` These are accessor functions for the `startAngle` and `endAngle` data fields. Note that because these values are real, these functions return their results on the top of the FPU stack. These functions return the angles in degrees.

`set_startAngle,`

`set_endAngle` These are the mutator functions for the `startAngle` and `endAngle` data fields. These functions expect the angle in degrees.

### 18.3.4.4 wPolygon\_t

A polygon is a closed geometric object created by drawing a set of lines between the points in a list (and from the last point to the first point to close the object).

Note: Windows automatically resizes most geographic objects you draw (e.g., rectangles and ellipses). It does not, however, resize a polygon if you change its bounding box. The HOWL polygon class, fortunately, contains extra code to resize a polygon if you change the width or height of the bounding box. Therefore, when using HOWL, your programs can treat polygons just like other geometric objects with respect to the `resize` method.

```
ptArray :pointer to w.POINT;  // w.POINT:[x:dword, y:dword]

wPolygon_t:
  class inherits( wFilledFrame_t );

  var
    align( 4 );
    wPolygon_private:
      record

        points          :ptArray;
        scaledPoints    :ptArray;
        nPoints         :uns32;
        origW           :dword;
        origH           :dword;

      endrecord;

  procedure create_wPolygon
  (
    wrName      :string;
    parent      :dword;
    x           :dword;
    y           :dword;
    width       :dword;
    height      :dword;
    lineColor   :dword;
    fillColor   :dword;
    bkgColor    :dword
  );  external;

  override method destroy;                external;
  override method processMessage;          external;

  override method set_width;               external;
```

```

        override method set_height;                external;
        override method resize;                   external;

        method set_points
        (
            nPoints :dword;
            points   :ptArray
        );    external;

        method get_points;      @returns( "eax" );    external;
        method get_nPoints;     @returns( "eax" );    external;

    endclass;

```

**points** This is the address of an array of w.POINT objects in memory (each element is 8 bits, a four-byte x-coordinate followed by a four byte y-coordinate value). This is a private data field that applications should not access or modify directly.

**scaledPoints** This is the address of an array of w.POINT objects in memory (each element is 8 bits, a four-byte x-coordinate followed by a four byte y-coordinate value). The Polygon class uses this array (rather than points) to draw the polygon if the polygon has been resized since it was created. This is a private data field that applications should not access or modify directly.

**nPoints** This is the number of points in the array pointed at by the points field. If this field is zero, then the points data field may contain an arbitrary value. This is a private data field that applications should not access or modify directly.

**origW** This field holds the original (created) width of the current polygon. The polygon class uses this value to determine if it has to scale the polygon along the x-axis because the polygon has been resized. This is a private data field that applications should not access or modify directly.

**origH** This field holds the original (created) height of the current polygon. The polygon class uses this value to determine if it has to scale the polygon along the y-axis because the polygon has been resized. This is a private data field that applications should not access or modify directly.

**create\_wPolygon** This is the constructor for the wPolygon\_t class. If you call this as a class procedure (e.g., "wPolygon\_t.create\_wPolygon") then this procedure will allocate storage for a new wPolygon\_t object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then create\_wPolygon will initialize that object in-place. Note that this constructor initializes nPoints to zero (and points to NULL). So immediately upon creation, the polygon has no vertexes and it will not draw anything on the form until you provide a list of points.

**wrName:** HOWL assigns this string to the \_name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

**parent:** this is the handle of the wView\_t or wForm\_t object on which the polygon will be drawn.

**x, y, width, height:** These arguments form a bounding box in which the polygon will be drawn.

**lineColor:** this is the RGB color value for the pen that HOWL will use to draw the outline of the polygon.

**fillColor:** this is the RGB color value for the brush that HOWL will use to paint the interior of the polygon.

**bkgColor:** this is the RGB color value for the brush that HOWL will use to paint the exterior of the polygon. If the H.O. byte of this color value contains \$FF, then HOWL will use the RGB value in the L.O. three bytes to fill the initial object window's

	background color but will not redraw the background thereafter (making the object's background transparent).
<code>processMessage</code>	This is a private method that applications must not call.
<code>destroy</code>	This is the destructor for the <code>wPolygon_t</code> class. Normally, applications do not call this destructor directly; instead, a container will call this destructor automatically when the container is destroyed. However, if you've created an independent (of any container) polygon object, you should call this destructor to free the resources it uses when you are done with the polygon.
<code>set_width,</code> <code>set_height,</code> <code>resize</code>	This are overridden methods from the <code>wFilledFrame_t</code> class. They handle scaling the polygon when you change the size of the polygon's bounding box. See the descriptions in <code>wFillFrame_t</code> for more details. Note that the these functions will usually make a copy of the points data pointed at by the <code>points</code> field and then set <code>scaledPoints</code> to point at this new data (note, however, that if you reset the size back to the original size, then these functions will deallocate the storage pointed at by the <code>scaledPoints</code> field).
<code>set_points</code>	<p>This is the mutator for the <code>points</code> and <code>nPoints</code> data fields.</p> <p><b>nPoints:</b> This argument specifies the number of points in the <code>points</code> array passed as the second argument.</p> <p><b>points:</b> this is a pointer to an array of <code>nPoints</code> <code>w.POINT</code> elements. The <code>set_points</code> method will make a copy of this data into internally allocated storage (on the heap) and store a pointer to the new data in the <code>points</code> field (this call also frees any storage previously in use by the <code>points</code> and <code>scaledPoints</code> fields).</p>
<code>get_nPoints</code>	This accessor returns the number of points in the polygon (the value of the <code>nPoints</code> field).
<code>get_points</code>	The <code>get_points</code> accessor function returns the value of the <code>points</code> or <code>scaledPoints</code> field. It returns a pointer to the <code>points</code> field if the current bounding box width and height of the polygon haven't changed since the last <code>set_points</code> call. This method returns <code>scaledPoints</code> if the polygon has been resized.

### 18.3.4.5 `wRectangle_t`

The `wRectangle_t` graphic object displays a (clickable) rectangle on a window or form.

```

wRectangle_t:
  class inherits( wFilledFrame_t );

  procedure create_wRectangle
  (
    wrName      :string;
    parent      :dword;
    x           :dword;
    y           :dword;
    width       :dword;
    height      :dword;
    lineColor   :dword;
    fillColor   :dword
  ); external;

  override method processMessage;          external;

endclass;
```

`create_wRectangle`

This is the constructor for the `wRectangle_t` class. If you call this as a class procedure (e.g., `wRectangle_t.create_wRectangle`) then this procedure will allocate storage for a new `wRectangle_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `wRectangle_t` will initialize that object in-place.

**wrName:** HOWL assigns this string to the `_name` data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

**parent:** this is the handle of the window object on which the ellipse will be drawn.

**x, y, width, height:** These arguments form a bounding box in which the rectangle will be drawn. If `width` and `height` are the same value, then the rectangle will form a square.

**lineColor:** this is the RGB color value for the pen that HOWL will use to draw the outline of the rectangle.

**fillColor:** this is the RGB color value for the brush that HOWL will use to paint the interior of the rectangle.

`processMessage` This is a private method that applications must not call.

### 18.3.4.6 wRoundRect\_t

`wRoundRect_t` objects are graphic objects that are rectangles with rounded corners.

```
wRoundRect_t:
  class inherits( wFilledFrame_t );

  var
    align( 4 );
    wRoundRect_private:
      record

        cornerWidth      :dword;
        cornerHeight     :dword;

      endrecord;

  procedure create_wRoundRect
  (
    wrName      :string;
    parent      :dword;
    x           :dword;
    y           :dword;
    width       :dword;
    height      :dword;
    cornerWidth :dword;
    cornerHeight:dword;
    lineColor   :dword;
    fillColor   :dword;
    bkgColor    :dword
  ); external;

  method get_cornerWidth;      @returns( "eax" ); external;
  method get_cornerHeight;     @returns( "eax" ); external;

  method set_cornerWidth( cornerWidth:dword ); external;
  method set_cornerHeight( cornerHeight:dword ); external;

  override method processMessage; external;
```



```
endclass;
```

**cornerWidth** This data field controls the width of the ellipse that Windows draws on each corner of the rounded rectangle. Applications should not access this field directly, they should use the appropriate accessor and mutator functions to access or set the value of this data field. This value should be less than 1/2 the height of the round rectangle object.

**cornerHeight** This data field controls the height of the ellipse that Windows draws on each corner of the rounded rectangle. Applications should not access this field directly, they should use the appropriate accessor and mutator functions to access or set the value of this data field. This value should be less than 1/2 the height of the round rectangle object.

```
create_wRoundRect
```

This is the constructor for the `wRoundRect_t` class. If you call this as a class procedure (e.g., "`wRoundRect_t.create_wRoundRect`") then this procedure will allocate storage for a new `wRoundRect_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wRoundRect` will initialize that object in-place.

**wrName:** HOWL assigns this string to the `_name` data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

**parent:** this is the handle of the `wView_t` or `wForm_t` object on which the round rectangle will be drawn.

**x, y, width, height:** These arguments form a bounding box in which the round rectangle will be drawn.

**lineColor:** this is the RGB color value for the pen that HOWL will use to draw the outline of the round rectangle.

**fillColor:** this is the RGB color value for the brush that HOWL will use to paint the interior of the round rectangle.

**bkgColor:** this is the RGB color value for the brush that HOWL will use to paint the exterior of the round rectangle (the area just outside the rounded corners). If the H.O. byte of this color value contains \$FF, then HOWL will use the RGB value in the L.O. three bytes to fill the initial object window's background color but will not redraw the background thereafter (making the object's background transparent).

```
get_cornerWidth,
```

`get_cornerHeight` These are the accessor functions for the `cornerWidth` and `cornerHeight` data fields. Applications should call these methods rather than accessing the data fields directly.

```
set_cornerWidth,
```

`set_cornerHeight` These are the mutator functions for the `cornerWidth` and `cornerHeight` data fields. Applications should call these methods rather than writing directly to the data fields.

`processMessage` This is a private method that applications must not call.

## 18.3.5 Buttons

The HOWL button widgets come in two basic varieties: checkable (check boxes and radio buttons) and non-checkable (push buttons). All buttons are derived from the `wClickable_t` class. The constructors for these buttons let you initialize the `onClick` widgetProc associated with all buttons; you can also call the `set_onDblClick` method to make a button double-clickable.

## 18.3.6 wCheckBox\_t

`wCheckBox_t` objects have a binary state (checked or unchecked). Whenever the user clicks on a checkbox, the widget toggles its state. Note that `wCheckBox_t` objects inherit the fields of the `wCheckable_t` class. You can call the `get_check` and `set_check` methods of that class to get the current `wCheckBox_t` object state or to set it.

```

wCheckBox_t:
  class inherits( wCheckable_t );

  procedure create_wCheckBox
  (
    wcbName      :string;
    caption      :string;
    parent       :dword;
    x            :dword;
    y            :dword;
    width        :dword;
    height       :dword;
    onClick      :widgetProc
  ); external;

endclass;

create_wCheckBox

```

This is the constructor for the `wCheckBox_t` class. If you call this as a class procedure (e.g., "`wCheckBox_t.create_wCheckBox`") then this procedure will allocate storage for a new `wCheckBox_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wCheckBox` will initialize that object in-place.

**wcbName:** HOWL assigns this string to the `_name` data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

**caption:** this is the caption text that will be drawn immediately to the right of the check box. Windows makes an internal copy of this string, so the value need only exist for as long as the constructor call is in progress.

**parent:** this is the handle of the `wView_t` or `wForm_t` object on which the check box will be drawn.

**x, y, width, height:** These arguments form a bounding box in which the check box and caption will be drawn.

**onClick:** this is the name of a `widgetProc` that HOWL will call whenever you click on the checkbox widget. If this field contains NULL, HOWL will not call any `widgetProc` procedure.

### 18.3.7 wCheckBox3\_t

`wCheckBox3_t` checkboxes are similar to standard checkboxes except they have three states: checked, unchecked, and grayed. The `get_state` method (inherited from `wCheckable_t`) will return 0 (unchecked), 1 (checked), or 2 (grayed).

```

wCheckBox3_t:
  class inherits( wCheckable_t );

  procedure create_wCheckBox3
  (
    wcb3Name     :string;
    caption      :string;
    parent       :dword;
    x            :dword;
    y            :dword;
    width        :dword;
    height       :dword;
    onClick      :widgetProc
  ); external;

```

```

endclass;

create_wCheckBox3

```

This is the constructor for the `wCheckBox3_t` class. If you call this as a class procedure (e.g., `wCheckBox3_t.create_wCheckBox3`) then this procedure will allocate storage for a new `wCheckBox3_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wCheckBox3` will initialize that object in-place.

**wcbName:** HOWL assigns this string to the `_name` data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

**caption:** this is the caption text that will be drawn immediately to the right of the check box. Windows makes an internal copy of this string, so the value need only exist for as long as the constructor call is in progress.

**parent:** this is the handle of the `wView_t` or `wForm_t` object on which the check box will be drawn.

**x, y, width, height:** These arguments form a bounding box in which the check box and caption will be drawn.

**onClick:** this is the name of a `widgetProc` that HOWL will call whenever you click on the checkbox widget. If this field contains NULL, HOWL will not call any `widgetProc` procedure.

### 18.3.8 wCheckBox3LT\_t

`wCheckBox3LT_t` checkboxes are similar to `wCheckBox3_t` checkboxes except they draw the caption text to the left of the checkbox rather than to the right of it. The `get_state` method (inherited from `wCheckable_t`) will return 0 (unchecked), 1 (checked), or 2 (grayed).

```

wCheckBox3LT_t:
    class inherits( wCheckable_t );

    procedure create_wCheckBox3LT
    (
        wcb3ltName    :string;
        caption       :string;
        parent        :dword;
        x             :dword;
        y             :dword;
        width         :dword;
        height        :dword;
        onClick       :widgetProc
    );    external;

endclass;

create_wCheckBox3LT

```

This is the constructor for the `wCheckBox3LT_t` class. If you call this as a class procedure (e.g., `wCheckBox3LT_t.create_wCheckBox3LT`) then this procedure will allocate storage for a new `wCheckBox3LT_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wCheckBox3LT` will initialize that object in-place.

**wcbName:** HOWL assigns this string to the `_name` data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

**caption:** this is the caption text that will be drawn immediately to the left of the check box. Windows makes an internal copy of this string, so the value need only exist for as long as the constructor call is in progress.

**parent:** this is the handle of the `wView_t` or `wForm_t` object on which the check box will be drawn.

**x, y, width, height:** These arguments form a bounding box in which the check box and caption will be drawn.

**onClick:** this is the name of a `widgetProc` that HOWL will call whenever you click on the checkbox widget. If this field contains NULL, HOWL will not call any `widgetProc` procedure.

### 18.3.9 wCheckBoxLT\_t

`wCheckBoxLT_t` checkboxes are similar to `wCheckBox_t` checkboxes except they draw the caption text to the left of the checkbox rather than to the right of it.

```
wCheckBoxLT_t:
    class inherits( wCheckable_t );

    procedure create_wCheckBoxLT
    (
        wcb3ltName    :string;
        caption       :string;
        parent        :dword;
        x             :dword;
        y             :dword;
        width         :dword;
        height        :dword;
        onClick       :widgetProc
    );    external;

endclass;

create_wCheckBoxLT
```

This is the constructor for the `wCheckBoxLT_t` class. If you call this as a class procedure (e.g., "`wCheckBoxLT_t.create_wCheckBoxLT`") then this procedure will allocate storage for a new `wCheckBoxLT_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wCheckBoxLT` will initialize that object in-place.

**wcbName:** HOWL assigns this string to the `_name` data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

**caption:** this is the caption text that will be drawn immediately to the left of the check box. Windows makes an internal copy of this string, so the value need only exist for as long as the constructor call is in progress.

**parent:** this is the handle of the `wView_t` or `wForm_t` object on which the check box will be drawn.

**x, y, width, height:** These arguments form a bounding box in which the check box and caption will be drawn.

**onClick:** this is the name of a `widgetProc` that HOWL will call whenever you click on the checkbox widget. If this field contains NULL, HOWL will not call any `widgetProc` procedure.

### 18.3.10 wPushButton\_t

wPushButton\_t objects are standard Windows push button widgets. They almost always invoke some sort of "onClick" widgetProc procedure when the button is pressed.

```
wPushButton_t:
    class inherits( wButton_t );

    procedure create_wPushButton
    (
        wpbName      :string;
        caption      :string;
        parent       :dword;
        x            :dword;
        y            :dword;
        width        :dword;
        height       :dword;
        onClick      :widgetProc
    ); external;

endclass;

create_wPushButton
```

This is the constructor for the wPushButton\_t class. If you call this as a class procedure (e.g., "wPushButton\_t.create\_wPushButton") then this procedure will allocate storage for a new wPushButton\_t object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then create\_wPushButton will initialize that object in-place.

**wcbName:** HOWL assigns this string to the \_name data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

**caption:** this is the caption text that will be drawn on the push button. Windows makes an internal copy of this string, so the value need only exist for as long as the constructor call is in progress.

**parent:** this is the handle of the wView\_t or wForm\_t object on which the push button will be drawn.

**x, y, width, height:** These arguments form a bounding box in which the push button and caption will be drawn.

**onClick:** this is the name of a widgetProc that HOWL will call whenever you click on the push button widget. If this field contains NULL, HOWL will not call any widgetProc procedure.

### 18.3.11 wRadioButton\_t

wRadioButton\_t objects are stand-alone radio buttons on a form. You'll rarely use these objects because radio buttons are generally employed in sets (using a wRadioSet\_t container and wRadioSetButton\_t objects). A stand-alone radio button is essentially a check box with a circle and a dot rather than a square and an "x". The main purpose for wRadioButton\_t objects (and wRadioButtonLT\_t objects) is for programmers who want to manually control the operation of the radio buttons.

```
wRadioButton_t:
    class inherits( wCheckable_t );

    procedure create_wRadioButton
    (
        wrbName      :string;
        caption      :string;
```

```

        parent      :dword;
        x           :dword;
        y           :dword;
        width       :dword;
        height      :dword;
        onClick     :widgetProc
    );    external;

```

```
endclass;
```

```
create_wRadioButton
```

This is the constructor for the `wRadioButton_t` class. If you call this as a class procedure (e.g., `"wRadioButton_t.create_wRadioButton"`) then this procedure will allocate storage for a new `wRadioButton_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wRadioButton` will initialize that object in-place.

**wrbName:** HOWL assigns this string to the `_name` data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

**caption:** this is the caption text that will be drawn to the right of the radio button. Windows makes an internal copy of this string, so the value need only exist for as long as the constructor call is in progress.

**parent:** this is the handle of the `wView_t` or `wForm_t` object on which the radio button will be drawn.

**x, y, width, height:** These arguments form a bounding box in which the radio button and caption will be drawn.

**onClick:** this is the name of a `widgetProc` that HOWL will call whenever you click on the radio button widget. If this field contains `NULL`, HOWL will not call any `widgetProc` procedure. Generally, if you're using `wRadioButton_t` objects in your application, it is the responsibility of the `onClick` procedure to properly update the other radio buttons associated with the one the user has just clicked on.

### 18.3.12 wRadioButtonLT\_t

`wRadioButton_LT` objects are just like `wRadioButton_t` objects except the text appears to the left of the radio button rather than to the right.

```

wRadioButtonLT_t:
    class inherits( wCheckable_t );

    procedure create_wRadioButtonLT
    (
        wrbltName    :string;
        caption      :string;
        parent       :dword;
        x            :dword;
        y            :dword;
        width        :dword;
        height       :dword;
        onClick      :widgetProc
    );    external;

endclass;

```

```
create_wRadioButtonLT
```

This is the constructor for the `wRadioButton_t` class. If you call this as a class procedure (e.g., `"wRadioButtonLT_t.create_wRadioButtonLT"`) then this procedure will allocate storage for a new `wRadioButtonLT_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wRadioButtonLT` will initialize that object in-place.

**wrbtName:** HOWL assigns this string to the `_name` data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

**caption:** this is the caption text that will be drawn to the left of the radio button. Windows makes an internal copy of this string, so the value need only exist for as long as the constructor call is in progress.

**parent:** this is the handle of the `wView_t` or `wForm_t` object on which the radio button will be drawn.

**x, y, width, height:** These arguments form a bounding box in which the radio button and caption will be drawn.

**onClick:** this is the name of a `widgetProc` that HOWL will call whenever you click on the radio button widget. If this field contains `NULL`, HOWL will not call any `widgetProc` procedure. Generally, if you're using `wRadioButtonLT_t` objects in your application, it is the responsibility of the `onClick` procedure to properly update the other radio buttons associated with the one the user has just clicked on. Note that the `wRadioSet..endwRadioSet` statement in the HOWL declarative language will report an error if you attempt to add some non-radio-set-button widget to the `wRadioSet_t` object you're creating.

### 18.3.13 wRadioSet\_t

A `wRadioSet_t` object is a container that holds (only) `wRadioSetButton_t` and `wRadioSetButtonLT_t` objects. The `wRadioSet_t` object automatically maintains all the buttons it contains, ensuring that (at most) one button is checked at a time. Note that an application must only insert groups of `wRadioSetButton_t` and `wRadioSetButtonLT_t` objects into the widget list of a `wRadioSet_t` object. If an application (manually) inserts other objects into a `wRadioSet_t` widget list, the radio buttons may not behave properly. Visually, a `wRadioSet_t` object is identical to a `wGroupBox_t` object. That is, it is a rectangular panel with a caption in the upper-left-hand corner of the rectangle.

```
wRadioSet_t:
    class inherits( wContainer_t );

    var
        align( 4 );
        wRadioSet_private:
            record

                // Windows handle for the group box window

                groupBoxHndl      :dword;

            endrecord;

    procedure create_wRadioSet
    (
        wrsName      :string;
        caption      :string;
        parent       :dword;
        x            :dword;
        y            :dword;
```

```

        width      :dword;
        height     :dword;
        bkgColor   :dword
    );    external;

    override method processMessage;          external;
    override method destroy;                 external;
    override method set_width;               external;
    override method set_height;              external;
    override method resize;                  external;

endclass;

groupBoxHndl    This is the handle for the actual group box (a separate surface for the background is use to
                  fill in the area behind the group box). This is a private field. Applications should not
                  access it.

create_wRadioSet This is the constructor for the wRadioSet_t class. If you call this as a class procedure (e.g.,
                  "wRadioSet_t.create_wRadioSet") then this procedure will allocate storage for a new
                  wRadioSet_t object on the heap and return a pointer to that object in ESI. If you make a
                  standard object call to this constructor, then create_wRadioSet will initialize that
                  object in-place.

wrsName: HOWL assigns this string to the _name data field of the object. This string's
value should be constant over the execution lifetime of the newly initialized object.

caption: this is the caption text that will be drawn in the upper-left-hand corner of the
wRadioSet_t's panel rectangle. Windows makes an internal copy of this string, so the
value need only exist for as long as the constructor call is in progress.

parent: this is the handle of the wView_t or wForm_t object on which the radio set group
box will be drawn.

x, y, width, height: These arguments form a bounding box in which the radio button and
caption will be drawn.

processMessage   This is a private method. Applications must not call it.

destroy          This is the class constructor. Usually, a container object will call this destructor
                  automatically for you; applications don't normally call this destructor unless they create a
                  wRadioSet_t object and don't insert it into some container's widget list.

set_width,
set_height,
resize          These fields are overridden from the wVisual_t class. See the description there for more
                  details.

```

You will want to call the `insertWidget` method (inherited from `wContainer_t`) in order to add `wRadioSetButton_t` or `wRadioSetButtonLT_t` objects to a `wRadioSet_t` object.

### 18.3.13.1 wRadioSetButton\_t

A `wRadioSetButton_t` is a standard radio set button that appears within a `wRadioSet_t` group box. `wRadioSetButton_t` objects are identical to `wRadioButton_t` objects except that they support automatic radio button control on a `wRadioSet_t` group box. You can actually specify `wRadioSetButton_t` objects outside of a `wRadioSet_t` group box; however, HOWL will only maintain automatic radio button operation on those buttons you declare (in the HOWL declarative language) in a sequence without any other intervening widget types (except `wRadioSetButtonLT_t` objects, which can be intermixed with `wRadioSetButton_t` objects).

```

wRadioSetButton_t:
    class inherits( wCheckable_t );

```



```

procedure create_wRadioSetButton
(
    wrbName      :string;
    caption      :string;
    style        :dword;
    parent       :dword;
    x            :dword;
    y            :dword;
    width        :dword;
    height       :dword;
    onClick      :widgetProc
); external;

```

```
endclass;
```

```
create_wRadioSetButton
```

This is the constructor for the `wRadioSetButton_t` class. If you call this as a class procedure (e.g., "`wRadioSetButton_t.create_wRadioSetButton`") then this procedure will allocate storage for a new `wRadioSetButton_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wRadioSetButton` will initialize that object in-place.

**wrbName:** HOWL assigns this string to the `_name` data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

**caption:** this is the caption text that will be drawn to the right of the radio button. Windows makes an internal copy of this string, so the value need only exist for as long as the constructor call is in progress.

**parent:** this is the handle of the `wView_t` or `wForm_t` object on which the radio button will be drawn.

**x, y, width, height:** These arguments form a bounding box in which the radio button and caption will be drawn.

**onClick:** this is the name of a `widgetProc` that HOWL will call whenever you click on the radio button widget. If this field contains `NULL`, HOWL will not call any `widgetProc` procedure. Generally, if you're using `wRadioButton_t` objects in your application, it is the responsibility of the `onClick` procedure to properly update the other radio buttons associated with the one the user has just clicked on.

### 18.3.13.2 wRadioSetButtonLT\_t

A `wRadioSetButtonLT_t` is identical to a `wRadioSetButton_t` object except it draws the caption text to the left of the button rather than to the right of the button.

```

wRadioSetButtonLT_t:
    class inherits( wCheckable_t );

    procedure create_wRadioSetButtonLT
    (
        wrbltName  :string;
        caption    :string;
        style      :dword;
        parent     :dword;
        x          :dword;
        y          :dword;
        width      :dword;
        height     :dword;
        onClick    :widgetProc
    )

```

```

        );    external;

    endclass;

create_wRadioSetButtonLT

```

This is the constructor for the `wRadioSetButtonLT_t` class. If you call this as a class procedure (e.g., `"wRadioSetButtonLT_t.create_wRadioSetButtonLT"`) then this procedure will allocate storage for a new `wRadioSetButton_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wRadioSetButtonLT` will initialize that object in-place.

**wrbName:** HOWL assigns this string to the `_name` data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

**caption:** this is the caption text that will be drawn to the left of the radio button. Windows makes an internal copy of this string, so the value need only exist for as long as the constructor call is in progress.

**parent:** this is the handle of the `wView_t` or `wForm_t` object on which the radio button will be drawn.

**x, y, width, height:** These arguments form a bounding box in which the radio button and caption will be drawn.

**onClick:** this is the name of a `widgetProc` that HOWL will call whenever you click on the radio button widget. If this field contains `NULL`, HOWL will not call any `widgetProc` procedure. Generally, if you're using `wRadioButton_t` objects in your application, it is the responsibility of the `onClick` procedure to properly update the other radio buttons associated with the one the user has just clicked on.

## 18.3.14 Editors and Edit Boxes

The HOWL edit widgets allow users to enter passwords, single lines of text, or text documents (up to 32KB long). Users can cut and paste data between edit widgets and perform many other text-editing functions. Applications can select text from an edit widget, insert text into the widget, or extract text from the widget. Indeed, with just a little extra code, it's quite possible to create a fully-featured text editor using the HOWL edit widgets.

Perhaps the biggest limitation to these widgets is their 32K text limitation. A future version of HOWL will include an extended text editor that overcomes this limitation.

All of the HOWL edit widgets are subclasses of the `wabsEditBox_t` class and, therefore, inherit all the fields and methods from that abstract base class. You should take a moment to review that abstract base class before looking at the following derived class definitions.

### 18.3.14.1 wEditBox\_t

A `wEditBox_t` object allows a user to enter a single string (a single line of text) from the keyboard.

```

wEditBox_t:
    class inherits( wabsEditBox_t );

    procedure create_wEditBox
    (
        webName      :string;
        initialTxt    :string;
        parent        :dword;
        x             :dword;
        y             :dword;
        width         :dword;
        height        :dword;
        style         :dword;
    )

```

```

        onChange      :widgetProc
    );  external;

endclass;

```

**create\_wEditBox** This is the constructor for the `wEditBox_t` class. If you call this as a class procedure (e.g., "`wEditBox_t.create_wEditBox`") then this procedure will allocate storage for a new `wEditBox_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wEditBox` will initialize that object in-place.

**webName:** HOWL assigns this string to the `_name` data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

**initialTxt:** HOWL will initialize the edit box's text entry field with this string. This is commonly an empty string in most objects.

**parent:** this is the handle of the `wView_t` or `wForm_t` object on which the edit box will be drawn.

**x, y, width, height:** These arguments form a bounding box in which the edit box will be drawn.

**style:** This argument is zero or more of the Windows edit box styles logically OR'd together (or zero, to use the default edit box style). See the discussion of Windows edit box styles in the section on the `wEditBox` object earlier in this document (in the HOWL declaration language description).

**onChange:** this is the name of a `widgetProc` that HOWL will call whenever you change any text in the edit box widget. If this field contains NULL, HOWL will not call any `widgetProc` procedure. Generally, this field will contain NULL and you will process the text in an edit box in response to some other system event (such as a button press or loss of focus).

### 18.3.14.2 wPasswdBox\_t

A `wPasswdBox_t` object is almost identical to a `wEditBox_t` object. The difference is that Windows substitutes asterisks (or some other user-defined character) for the characters the user types at the keyboard to protect passwords from prying eyes.

```

wPasswdBox_t:
    class inherits( wabsEditBox_t );

    procedure create_wPasswdBox
    (
        wpwbName      :string;
        initialTxt     :string;
        parent         :dword;
        x              :dword;
        y              :dword;
        width          :dword;
        height         :dword;
        style           :dword;
        onChange       :widgetProc
    );  external;

    method get_passwordChar; @returns( "eax" ); external;
    method set_passwordChar( pwc:char );      external;

endclass;

```

```
create_wPasswdBox
```

This is the constructor for the `wPasswdBox_t` class. If you call this as a class procedure (e.g., `wPasswdBox_t.create_wPasswdBox`) then this procedure will allocate storage for a new `wPasswdBox_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wPasswdBox` will initialize that object in-place.

**webName:** HOWL assigns this string to the `_name` data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

**initialTxt:** HOWL will initialize the password box's text entry field with this string. This is commonly an empty string in most objects.

**parent:** this is the handle of the `wView_t` or `wForm_t` object on which the password box will be drawn.

**x, y, width, height:** These arguments form a bounding box in which the password box will be drawn.

**style:** This argument is zero or more of the Windows edit box styles logically OR'd together (or zero, to use the default edit box style). See the discussion of Windows edit box styles in the section on the `wEditBox` object earlier in this document (in the HOWL declaration language description).

**onChange:** this is the name of a `widgetProc` that HOWL will call whenever you change any text in the password box widget. If this field contains NULL, HOWL will not call any `widgetProc` procedure. Generally, this field will contain NULL and you will process the text in a password box in response to some other system event (such as a button press or loss of focus).

### 18.3.14.3 wTextEdit\_t

A `wTextEdit_t` object allows the user to enter multiple lines of text in a text editor format. If the text editor string data contains more lines (or more characters on a given line) than will fit in the text editor window, Windows will automatically attach scroll bars to the window so the user can scroll through the text data.

```
wTextEdit_t:
    class inherits( wabsEditBox_t );

    procedure create_wTextEdit
    (
        wteName      :string;
        initialTxt    :string;
        parent        :dword;
        x             :dword;
        y             :dword;
        width         :dword;
        height        :dword;
        style         :dword;
        onChange      :widgetProc
    );    external;

    method getLineCount; @returns( "eax" );           external;

    method getLineIndex( charIndex:dword );
        @returns( "eax" );                           external;

    method getCharIndex( lineIndex:dword );
        @returns( "eax" );                           external;
```

```

method getLine( lineIndex:dword; txt:string );           external;

method a_getLine( lineIndex:dword );
    @returns( "eax" );                                   external;

method scroll( horz:int32; vert:int32 );                 external;
method scrollCaret;                                     external;
method setTabStops( tabstops:dword );                 external;

endclass;

create_wTextEdit

```

This is the constructor for the `wTextEdit_t` class. If you call this as a class procedure (e.g., `"wTextEdit_t.create_wTextEdit"`) then this procedure will allocate storage for a new `wTextEdit_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wTextEdit` will initialize that object in-place.

**webName:** HOWL assigns this string to the `_name` data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

**initialTxt:** HOWL will initialize the text editor's text entry field with this string. This is commonly an empty string in most objects.

**parent:** this is the handle of the `wView_t` or `wForm_t` object on which the text editor will be drawn.

**x, y, width, height:** These arguments form a bounding box in which the text editor will be drawn.

**style:** This argument is zero or more of the Windows edit box styles logically OR'd together (or zero, to use the default edit box style). See the discussion of Windows edit box styles in the section on the `wEditBox` object earlier in this document (in the HOWL declaration language description).

**onChange:** this is the name of a `widgetProc` that HOWL will call whenever you change any text in the text editor widget. If this field contains NULL, HOWL will not call any `widgetProc` procedure. Generally, this field will contain NULL and you will process the text in a text editor widget in response to some other system event (such as a button press or loss of focus).

<code>get_lineCount</code>	This method returns the number of lines of text in the text editor widget.
<code>get_lineIndex</code>	Given a zero-based character index into the text editor's string (up to 32K), this function will return a zero-based line number for that index (that is, the line number of the line that contains that particular character).
<code>get_charIndex</code>	Given a (zero-based) line number into the text editor's data string, this function returns the (zero-based) character index (into the text editor's string data) of the first character on that line.
<code>get_line</code>	Given a line index into the text editor's data string, this function returns the specified string.
	<b>lineIndex</b> This is the zero-based line index into the text editor's data string.
	<b>txt:</b> this is a string object where <code>get_line</code> will copy the string data for the specified line. This string must be previously allocated and have sufficient storage to hold the string or HOWL will raise an exception.
<code>a_get_line</code>	Given a line index, this method makes a copy of the specified text editor line on the heap and returns a pointer to this string in the EAX register. It is the caller's responsibility to free the storage associated with this string when the application is done using it.
<code>scroll</code>	This function will scroll the text editor window the number of characters specified by the two arguments in the horizontal and vertical directions.

**horz:** this is the number of characters to scroll in the horizontal direction.

	<b>vert:</b> this is the number of characters to scroll in the vertical direction. Windows will not let you scroll beyond the last line in the text editor's string; if you attempt to do so, Windows will simply display the last line of text at the top of the editor's window.
<code>scrollCaret</code>	Positions the text display window so that the text containing the insertion caret is visible on the screen.
<code>setTabStops</code>	This function sets tabstops every 'tabstops' characters, where 'tabstops' is the argument you pass to this method.

### 18.3.15 List, Drag, and Combo Boxes

List and drag boxes are tables of data from which the user can select an item (a row) by clicking on the line of text associated with that item. Applications can add, delete, or rearrange lines of text in list and drag boxes. End users can rearrange data in a drag box with no interaction from the application.

#### 18.3.15.1 `wListBox_t`

A `wListBox_t` object is a table of strings created by the application. The user can click or double-click on these strings. The application can insert and delete strings in the list box. If there are too many strings to display in the window, then Windows will attach a vertical scroll bar to the list box and allow the user to scroll through the list box entry.

A list box will either display the strings in the order the application inserts them into the list box, or it can display them in a sorted order. You specify whether you want a sorted or unsorted list box when you create it.

```

wListBox_t:
  class inherits ( wClickable_t );

  var
    align( 4 );
    wListBox_private:
      record

        textColor    :dword;

      endrecord;

  procedure create_wListBox
  (
    wlbName      :string;
    parent       :dword;
    x            :dword;
    y            :dword;
    width        :dword;
    height       :dword;
    sort         :boolean;
    onClick      :widgetProc
  ); external;

  method add_string( s:string );           external;
  method insert_string( index:dword; s:string ); external;
  method delete_string( index:dword );     external;
  method reset;                           external;

  method find_prefix
  (
    s            :string;
    startIndex   :dword
  ); @returns( "eax" );                    external;

  method find_string

```

```

(
    s          :string;
    startIndex :dword
); @returns( "eax" ); external;

method get_count;          @returns( "eax" ); external;
method get_curSel;        @returns( "eax" ); external;
method get_itemData( i:dword ); @returns( "eax" ); external;
method a_get_text( i:dword ); @returns( "eax" ); external;
method get_text( i:dword; s:string ); external;

method set_curSel(index:dword); @returns( "eax" ); external;
method set_itemData
(
    index :dword;
    data  :dword
); external;

method load_dir
(
    pathname :string;
    attributes :dword
); external;

method get_textColor; @returns( "eax" ); external;
method set_textColor( textColor:dword ); external;

override method processMessage; external;

endclass;

```

**textColor** This is the RGB color that Windows will use to draw the text on the listbox. This is a private data field; applications should only access this value using the associated access and mutator. The constructor initializes the text color to black.

**create\_wListBox**

This is the constructor for the `wListBox_t` class. If you call this as a class procedure (e.g., `wListBox_t.create_wListBox`) then this procedure will allocate storage for a new `wListBox_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wListBox` will initialize that object in-place.

**wlbName:** HOWL assigns this string to the `_name` data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

**parent:** this is the handle of the `wView_t` or `wForm_t` object on which the list box will be drawn.

**x, y, width, height:** These arguments form a bounding box in which the list box will be drawn.

**sort:** if this field is true, then the constructor will create a sorted list box. If this value is false, then the constructor will create an unsorted list box.

**onClick:** this is the name of a `widgetProc` that HOWL will call whenever select a (new) line in a list box.

**add\_string**

This method appends a string (specified as the argument) to the end of an unsorted list box. It will insert the string at the proper position within a sorted list box. Note that Windows will make an internal copy of the string's data.

**insert\_string**

This method inserts a string before some other entry in the list box. This method ignores the sorted/unsorted state of the list box and always inserts the string at the specified index. The index must be in the range 0..count where count is the number of entries in the list

	<p>box. If you specify the value count as the index, then this method appends the item to the end of the list (similar to <code>add_string</code>, except no sorting).</p> <p><b>index:</b> the line number before which the string is to be inserted. This is a zero-based index.</p> <p><b>s:</b> this is the string data to insert into the list box.</p>
<code>delete_string</code>	This method deletes the string at the specified index in the list box.
<code>reset</code>	This method deletes all the strings in the list box.
<code>find_prefix</code>	<p>This method searches for a line of text in the list box that begins with some string. This method begins searching starting with an application-defined line index into the list box. This method returns the index into the list box where the string prefix was found, or the constant <code>w.LB_ERR</code> if it could find no string with the specified prefix.</p> <p><b>s:</b> the string prefix to search for in the list box.</p> <p><b>startIndex:</b> the starting line index to begin the search.</p>
<code>find_string</code>	<p>This method searches for a line of text in the list box that matches some string. This method begins searching starting with an application-defined line index into the list box. This method returns the index into the list box where the string was found, or the constant <code>w.LB_ERR</code> if it could not find the string.</p> <p><b>s:</b> the string to search for in the list box.</p> <p><b>startIndex:</b> the starting line index to begin the search.</p>
<code>get_count</code>	This method returns the number of lines in the list box (in EAX).
<code>get_curSel</code>	This method returns the index of the currently selected item in the list box (in EAX).
<code>get_itemData</code>	<p>Each line of text in a list box has a 32-bit user-defined data value associated with it. You could, for example store a pointer to some object or other data type in this field and retrieve it when the user selects an item in the list box. The <code>get_itemData</code> method retrieves this user data from the list box. The single argument is the index of the list box entry for which you want the user data (you would typically supply the data returned by <code>get_curSel</code> as this argument).</p>
<code>a_get_text</code>	This method makes a copy of the string data (on the heap) for the line in the list box at the index specified by the argument. It returns a pointer to this new string in EAX. It is the caller's responsibility to free the storage for this string when the caller is done with it.
<code>get_text</code>	<p>This method makes a copy of the string data for the line in the list box at a user-supplied index. It stores the string data into a string object whose address the caller passes as an argument. That string must have sufficient storage allocated for it or HOWL will raise an exception.</p> <p><b>i:</b> index of the line in the list box whose string data this method will extract.</p> <p><b>s:</b> pointer to a string object where this method will store the result.</p>
<code>set_curSel</code>	This method highlights the line at the specified index in the list box (it becomes the "currently selected" item).
<code>set_itemData</code>	<p>This method allows you to associate a user-defined 32-bit data value with an item in the list box. If 32 bits is insufficient for your needs, you can always store a pointer to the actual data in this data area.</p> <p><b>index:</b> this is the (zero-based) index of the line in the list box that you want to attach the data to.</p> <p><b>data:</b> this is the 32-bit value you want to associate with the line in the list box.</p>
<code>load_dir</code>	<p>This method populates the list box with the file names from the directory specified by the <code>pathname</code> argument.</p> <p><b>pathname:</b> an ambiguous pathname (e.g., "c:\*.*)" that specifies the path to the files and the files at that path that you want to load into the list box.</p> <p><b>attributes:</b> either zero, or the logical OR of one or more of the following Windows' attribute constants:</p>



w.DDL_ARCHIVE	Includes archived files.
w.DDL_DIRECTORY	Includes subdirectories. Subdirectory names are enclosed in square brackets ([ ]).
w.DDL_DRIVES	Includes drives. Drives are listed in the form [-x-], where x is the drive letter.
w.DDL_EXCLUSIVE	Includes only files with the specified attributes. By default, read-write files are listed even if DDL_READWRITE is not specified.
w.DDL_HIDDEN	Includes hidden files.
w.DDL_READONLY	Includes read-only files.
w.DDL_READWRITE	Includes read-write files with no additional attributes.
w.DDL_SYSTEM	Includes system files.

```

get_textColor,
set_textColor    These accessor/mutator functions get and set the text color that the widget uses.
processMessage    This is a private method. Applications must not call it.

```

### 18.3.15.2 wDragListBox\_t

A `wDragListBox_t` object is a special kind of list box that allows the user to rearrange the items in the list box without any interaction from the application. An application uses drag list boxes exactly like list boxes (except, of course, for the object's type name). As the `wDragListBox_t` class is derived from `wListBox_t`, all of the list box methods are available to `wDragListBox_t` objects. Note that HOWL does not offer drag list boxes the "sort" option, which makes little sense as the end user will probably rearrange their drag list boxes thus defeating the purpose of the sort option.

```

wDragListBox_t:
  class inherits ( wListBox_t );

  var
    align( 4 );
    wDragListBox_private:
      record

        // The following is a private field.
        // External code should not access it.

        startDragIndex :dword;

      endrecord;

  procedure create_wDragListBox
  (
    wlbName      :string;
    parent       :dword;
    x            :dword;
    y            :dword;
    width        :dword;
    height       :dword;
    onClick      :widgetProc
  ); external;

  override method processMessage;          external;

```

```
endclass;
```

```
create_wDragListBox
```

This is the constructor for the `wDragListBox_t` class. If you call this as a class procedure (e.g., `"wDragListBox_t.create_wDragListBox"`) then this procedure will allocate storage for a new `wDragListBox_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wDragListBox` will initialize that object in-place.

**wlbName:** HOWL assigns this string to the `_name` data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

**parent:** this is the handle of the `wView_t` or `wForm_t` object on which the drag list box will be drawn.

**x, y, width, height:** These arguments form a bounding box in which the drag list box will be drawn.

**onClick:** this is the name of a `widgetProc` that HOWL will call whenever the user selects a (new) line in a drag list box.

### 18.3.15.3 wComboBox\_t

The `wComboBox_t` object is a combination of an edit box, a list box, and a pull-down menu. The user can type text directly into a list box or click on a button attached to the combo box and select an item from a list box that appears in a pull-down menu.

```
wComboBox_t:
  class inherits ( wListBox_t );

  var
    align( 4 );
    wComboBox_private:
      record

        onEditChange      :widgetProc;
        onCancel          :widgetProc;
        onSelEndOk         :widgetProc;

      endrecord;

  procedure create_wComboBox
  (
    wcbName      :string;
    caption      :string;
    parent       :dword;
    x            :dword;
    y            :dword;
    width        :dword;
    height       :dword;
    sort         :boolean;
    onSelChange  :widgetProc
  );  external;

  method get_onEditChange;    @returns( "eax" );    external;
  method get_onCancel;       @returns( "eax" );    external;
  method get_onSelEndOk;      @returns( "eax" );    external;

  method set_onEditChange( onEditChange:widgetProc ); external;
  method set_onCancel( onCancel:widgetProc );      external;
```

```

method set_SelEndOk( onSelEndOk:widgetProc );           external;

method a_get_editBoxText;    @returns( "eax" );         external;
method get_editBoxText( theText:string );              external;

method set_editBoxText( theText:string );              external;

override method load_dir;                                external;
override method processMessage;                        external;
override method add_string;                            external;
override method insert_string;                        external;
override method delete_string;                       external;
override method reset;                                external;
override method find_prefix;                          external;
override method find_string;                         external;
override method get_count;                            external;
override method get_curSel;                          external;
override method get_itemData;                        external;
override method a_get_text;                          external;
override method get_text;                            external;
override method set_curSel;                          external;
override method set_itemData;                        external;

endclass;

```

onEditChange	This is a pointer to a widgetProc that HOWL will call whenever the user makes a change to the edit box component of a combo box. If this field is NULL, HOWL will ignore it. Note that applications should only access or modify this field using the associated accessor/mutator methods.
onCancel	This is a pointer to a widgetProc that HOWL will call whenever the user cancels a change to the edit box component of a combo box. If this field is NULL, HOWL will ignore it. Note that applications should only access or modify this field using the associated accessor/mutator methods.
onSelEndOk	This is a pointer to a widgetProc that HOWL will call whenever the user selects an item from the pull-down list box and the application should select that entry. If this field is NULL, HOWL will ignore it. Note that applications should only access or modify this field using the associated accessor/mutator methods.

#### create\_wComboBox

This is the constructor for the `wComboBox_t` class. If you call this as a class procedure (e.g., `"wComboBox_t.create_wComboBox"`) then this procedure will allocate storage for a new `wComboBox_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wComboBox` will initialize that object in-place.

**wcbName:** HOWL assigns this string to the `_name` data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

**caption:** This is the initial string data for the combo box's edit box field.

**parent:** this is the handle of the `wView_t` or `wForm_t` object on which the combo box will be drawn.

**x, y, width, height:** These arguments form a bounding box in which the combo box will be drawn.

**sort:** like list boxes, combo boxes offer the option of sorting the list for you. If this field is true, HOWL will create a sorted list in the combo box; if this field is false, the list in the combo box will be unsorted.

**onSelChange**: this is the name of a `widgetProc` that HOWL will call whenever the user selects a (new) line in the list of a combo box.

```
get_onEditChange,
get_onCancel,
get_onSelEndOk
```

These are the accessor functions for the combo box's data fields.

```
set_onEditChange,
set_onCancel,
set_onSelEndOk
```

These are the mutator functions for the combo box's data fields.

**a\_get\_editBox\_text** This method returns a copy of the string data currently held in the combo box's edit box component. HOWL allocates storage for this string and returns a pointer to the new string in the EAX register. It is the caller's responsibility to free the storage for this string.

**get\_editBox\_text** This method retrieves the string from the combo box and stores the string data in the string object passed as a parameter. The destination string must have sufficient storage or HOWL will raise an exception.

**set\_editBox\_text** This function replaces the combo box's string data with the string passed as an argument.

The remaining methods listed in the `wComboBox_t` declaration above are overridden methods from the `wListBox_t` class. The overriding occurs for internal technical reasons. To an application, these methods are used exactly like those in a list box. Please see the discussion in the list box section for more details on the operation of these methods.

## 18.3.16 Progress Bars

A progress bar is a bar graph that shows the progress of some lengthy operation during the execution of an application.

### 18.3.16.1 wProgressBar\_t

The `wProgressBar_t` type implements Windows progress bars in HOWL. A progress bar has three main attributes: a current position (the current "progress"), a minimum position, and a maximum position. When drawing a progress bar, Windows will create a horizontal bar graph and will fill the bar graph from the left to the right based on the current position, minimum, and maximum values.

```
wProgressBar_t:
    class inherits( wVisual_t );

    var
        align( 4 );
        wProgressBar_private:
            record

                position      :word;
                align( 4 );

                lowRange      :word;
                hiRange       :word;

            endrecord;

    procedure create_wProgressBar
    (
        wpbName      :string;
        parent        :dword;
        x             :dword;
        y             :dword;
        width         :dword;
```

```

        height      :dword
    );  external;

    method get_position; @returns( "eax" );          external;
    method set_position( position:word );          external;

    method get_lowRange; @returns( "eax" );          external;
    method get_hiRange;  @returns( "eax" );          external;
    method set_range( low:word; high:word );        external;

endclass;

```

position	These data field is the current position of the track bar. Applications should never access this field directly. Instead, they should use the accessor/mutator methods to read or write this field's data.
lowRange	This is the minimum value for the progress bar's position. It is usually zero, but you can set any minimum value you like as long as it is less than the value held in the <code>hiRange</code> data field. You should never set the value of the <code>position</code> data field to a value lower than the <code>lowRange</code> value. Applications should never access this field directly. Instead, they should use the accessor/mutator methods to read or write this field's data.
hiRange	This is the maximum value for the progress bar's position. You can set any maximum value you like as long as it is greater than the value held in the <code>lowRange</code> data field. You should never set the value of the <code>position</code> data field to a value higher than the <code>hiRange</code> value. Applications should never access this field directly. Instead, they should use the accessor/mutator methods to read or write this field's data.

#### create\_wProgressBar

This is the constructor for the `wProgressBar_t` class. If you call this as a class procedure (e.g., `"wProgressBar_t.create_wProgressBar"`) then this procedure will allocate storage for a new `wProgressBar_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wProgressBar` will initialize that object in-place.

**wpbName:** HOWL assigns this string to the `_name` data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

**parent:** this is the handle of the `wView_t` or `wForm_t` object on which the progress bar will be drawn.

**x, y, width, height:** These arguments form a bounding box in which the progress bar will be drawn.

```

get_position,
get_lowRange,
get_hiRange
set_position,
set_lowRange,
set_hiRange

```

These are the accessor methods for the class' data fields.

These are the mutator functions for the class' data fields. Note that changing any of these values (and in particular, changing the position value) will cause Windows to redraw the progress bar reflecting the new values.

## 18.3.17 Scroll Bars and Track Bars

Scroll bars and track bars are positional input devices. Applications generally use scroll bars to specify a position to view on a form (when the form's contents are too large to fit in the window and one time) whereas a

trackbar provides a generic numeric input device selectable by the position of the slider on the trackbar. Both widgets are available in horizontal and vertical orientations.

Note that Windows can automatically associate horizontal and vertical scroll bars with a window when you create that window. For window scrolling purposes, this is generally how you create and use scroll bars. However, you can create stand-alone scroll bars for use by your applications using the scrollbar widget.

### 18.3.17.1 wScrollBar\_t

The wScrollBar\_t class lets you create stand-alone scroll bars in your application. Scroll bars offer a large number of event notifications that tell you about various user interactions with the scroll bar.

Scroll bars in early versions of Windows were limited to 16-bit range and position values. For the most part, later versions of Windows extended all these values to 32 bits. However, one important pair of values, returned when tracking movements on scroll bars, is still limited to 16-bit values. Therefore, if you plan to make full use of the scroll bar's feature set and notifications, you need to limit yourself to using 16-bit ranges and positions. In general, this is not a severe limitations because there aren't enough pixels on the screen to provide a granularity of 16 bits, much less 32. However, just note that although you can set the low and high range values for a scroll bar to arbitrary 32-bit values, you should limit yourself to 16-bit values.

```
wScrollBar_t:
    class inherits( wVisual_t );

    var
        align( 4 );
        wScrollBar_private:
            record

                onChange           :widgetProc;
                onThumbPosn        :widgetProc;
                onThumbTrack       :widgetProc;
                onLineLeft         :widgetProc;
                onLineRight        :widgetProc;
                onLineDown         :widgetProc;
                onLineUp           :widgetProc;
                onEndScroll        :widgetProc;
                onPageDown         :widgetProc;
                onPageUp           :widgetProc;
                onPageLeft         :widgetProc;
                onPageRight        :widgetProc;
                onTop               :widgetProc;
                onBottom           :widgetProc;

                lineInc            :uns32;
                pageInc            :uns32;
                curPosn            :dword;
                info               :w.SCROLLINFO;
                textColor          :dword;

            endrecord;

    procedure create_wScrollBar
    (
        wtbName      :string;
        parent        :dword;
        x             :dword;
        y             :dword;
        width         :dword;
        height        :dword;
        style         :dword;
        onChange      :widgetProc
    ); external;

    override method enable;                                external;
```

```

override method disable;                external;
override method show;                   external;
override method hide;                   external;

method get_position; @returns( "eax" );   external;
method set_position( position:dword );    external;

method get_lowRange; @returns( "eax" );   external;
method get_hiRange;  @returns( "eax" );   external;
method set_range( low:dword; high:dword ); external;

method get_onChange;      @returns( "eax" );   external;
method get_onThumbPosn;   @returns( "eax" );   external;
method get_onThumbTrack;  @returns( "eax" );   external;
method get_onLineDown;    @returns( "eax" );   external;
method get_onLineUp;      @returns( "eax" );   external;
method get_onLineLeft;    @returns( "eax" );   external;
method get_onLineRight;   @returns( "eax" );   external;
method get_onEndScroll;   @returns( "eax" );   external;
method get_onPageDown;    @returns( "eax" );   external;
method get_onPageUp;      @returns( "eax" );   external;
method get_onPageLeft;    @returns( "eax" );   external;
method get_onPageRight;   @returns( "eax" );   external;
method get_onTop;         @returns( "eax" );   external;
method get_onBottom;      @returns( "eax" );   external;
method get_lineInc;       @returns( "eax" );   external;
method get_pageInc;       @returns( "eax" );   external;

method set_onChange( onChange:widgetProc ); external;
method set_onThumbPosn( onThumbPosn:widgetProc ); external;
method set_onThumbTrack( onThumbTrack:widgetProc ); external;
method set_onLineDown( onLineDown:widgetProc ); external;
method set_onLineUp( onLineUp:widgetProc ); external;
method set_onLineLeft( onLineLeft:widgetProc ); external;
method set_onLineRight( onLineRight:widgetProc ); external;
method set_onEndScroll( onEndScroll:widgetProc ); external;
method set_onPageDown( onPageDown:widgetProc ); external;
method set_onPageUp( onPageUp:widgetProc ); external;
method set_onPageLeft( onPageLeft:widgetProc ); external;
method set_onPageRight( onPageRight:widgetProc ); external;
method set_onTop( onTop:widgetProc ); external;
method set_onBottom( onBottom:widgetProc ); external;

method set_lineInc( lineInc:dword ); external;
method set_pageInc( pageInc:dword ); external;

method get_textColor; @returns( "eax" ); external;

method set_textColor( textColor:dword ); external;

override method processMessage; external;

endclass;

```

onChange

This field, if non-NULL, points at a widgetProc procedure that HOWL will call whenever the user changes the position of the thumb on the scroll bar (using any means to

	change the value). Upon entry into the <code>widgetProc</code> procedure, the H.O. word of the <code>wParam</code> argument contains the current scroll bar position.
<code>onThumbPosn</code>	This field, if non-NULL, points at a <code>widgetProc</code> procedure that HOWL will call whenever the user changes the position of the thumb on the scroll bar by dragging the thumb. HOWL calls this procedure after the user has released the mouse button while dragging the thumb control. Upon entry into the <code>widgetProc</code> procedure, the H.O. word of the <code>wParam</code> argument contains the current scroll bar position.
<code>onThumbTrack</code>	This field, if non-NULL, points at a <code>widgetProc</code> procedure that HOWL will call while the user is dragging the thumb around the scroll bar. An application can use this notification to dynamically adjust the screen during thumb movement. Upon entry into the <code>widgetProc</code> procedure, the H.O. word of the <code>wParam</code> argument contains the current scroll bar position. Note that this is the only notification that provides only a 16-bit value for the thumb position; if you don't need to use this notification, it is possible to obtain 32-bit values for the thumb position (from the other notification calls).
<code>onLineLeft</code>	This field, if non-NULL, points at a <code>widgetProc</code> procedure that HOWL will call whenever the user changes the position of the thumb on a horizontal scroll bar by pressing the left arrow on the scroll bar. Upon entry into the <code>widgetProc</code> procedure, the H.O. word of the <code>wParam</code> argument contains the current scroll bar position.
<code>onLineRight</code>	This field, if non-NULL, points at a <code>widgetProc</code> procedure that HOWL will call whenever the user changes the position of the thumb on a horizontal scroll bar by pressing the right arrow on the scroll bar. Upon entry into the <code>widgetProc</code> procedure, the H.O. word of the <code>wParam</code> argument contains the current scroll bar position.
<code>onLineDown</code>	This field, if non-NULL, points at a <code>widgetProc</code> procedure that HOWL will call whenever the user changes the position of the thumb on a vertical scroll bar by pressing the down arrow on the scroll bar. Upon entry into the <code>widgetProc</code> procedure, the H.O. word of the <code>wParam</code> argument contains the current scroll bar position.
<code>onLineUp</code>	This field, if non-NULL, points at a <code>widgetProc</code> procedure that HOWL will call whenever the user changes the position of the thumb on a vertical scroll bar by pressing the up arrow on the scroll bar. Upon entry into the <code>widgetProc</code> procedure, the H.O. word of the <code>wParam</code> argument contains the current scroll bar position.
<code>onEndScroll</code>	This field, if non-NULL, points at a <code>widgetProc</code> procedure that HOWL will call after any scroll operation is complete. Upon entry into the <code>widgetProc</code> procedure, the H.O. word of the <code>wParam</code> argument contains the current scroll bar position.
<code>onPageDown</code>	This field, if non-NULL, points at a <code>widgetProc</code> procedure that HOWL will call whenever the user changes the position of the thumb on a vertical scroll bar by clicking on the scroll bar between the thumb and the down arrow on the scroll bar. Upon entry into the <code>widgetProc</code> procedure, the H.O. word of the <code>wParam</code> argument contains the current scroll bar position.
<code>onPageUp</code>	This field, if non-NULL, points at a <code>widgetProc</code> procedure that HOWL will call whenever the user changes the position of the thumb on a vertical scroll bar by clicking on the scroll bar between the thumb and the up arrow on the scroll bar. Upon entry into the <code>widgetProc</code> procedure, the H.O. word of the <code>wParam</code> argument contains the current scroll bar position.
<code>onPageLeft</code>	This field, if non-NULL, points at a <code>widgetProc</code> procedure that HOWL will call whenever the user changes the position of the thumb on a horizontal scroll bar by clicking on the scroll bar between the thumb and the left arrow on the scroll bar. Upon entry into the <code>widgetProc</code> procedure, the H.O. word of the <code>wParam</code> argument contains the current scroll bar position.
<code>onPageRight</code>	This field, if non-NULL, points at a <code>widgetProc</code> procedure that HOWL will call whenever the user changes the position of the thumb on a horizontal scroll bar by clicking on the scroll bar between the thumb and the right arrow on the scroll bar. Upon entry into the <code>widgetProc</code> procedure, the H.O. word of the <code>wParam</code> argument contains the current scroll bar position.



<code>onTop</code>	This field, if non-NULL, points at a <code>widgetProc</code> procedure that HOWL will call whenever the user moves the thumb all the way to the top on a vertical scroll bar or all the way to the left on a horizontal scroll bar. Upon entry into the <code>widgetProc</code> procedure, the H.O. word of the <code>wParam</code> argument contains the current scroll bar position (which should be the maximum value).
<code>onBottom</code>	This field, if non-NULL, points at a <code>widgetProc</code> procedure that HOWL will call whenever the user moves the thumb all the way to the bottom on a vertical scroll bar or all the way to the right on a horizontal scroll bar. Upon entry into the <code>widgetProc</code> procedure, the H.O. word of the <code>wParam</code> argument contains the current scroll bar position (which should be the maximum value).
<code>lineInc</code>	This data field contains the number of positions that will be added to or subtracted from the scroll bar thumb's current position when the user clicks on one of the scroll bar's arrow buttons. Applications should not access this data field directly, they should use the standard class accessor/mutator functions to read and write this value.
<code>pageInc</code>	This data field contains the number of positions that will be added to or subtracted from the scroll bar thumb's current position when the user clicks on the scroll bar between the thumb and one of the arrow buttons. Applications should not access this data field directly, they should use the standard class accessor/mutator functions to read and write this value.
<code>curPosn</code>	This is a private field used by the class. Applications should not access it.
<code>info</code>	This is a private field used by the class. Applications should not access it.
<code>textColor</code>	This is the RGB color that Windows will use to draw the text on the scrollbar. This is a private data field; applications should only access this value using the associated access and mutator. The constructor initializes the text color to black.
<code>bkgColor</code>	This is the RGB color that Windows will use to paint the background of the scrollbar. This is a private data field; applications should only access this value using the associated access and mutator. The constructor initializes the background color to white.
<code>bkgBrush</code>	This is a private data field; applications should never access it.

#### `create_wScrollBar`

This is the constructor for the `wScrollBar_t` class. If you call this as a class procedure (e.g., "`wScrollBar_t.create_wScrollBar`") then this procedure will allocate storage for a new `wScrollBar_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wScrollBar` will initialize that object in-place.

**wtbName:** HOWL assigns this string to the `_name` data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

**parent:** this is the handle of the `wView_t` or `wForm_t` object on which the scroll bar will be drawn.

**x, y, width, height:** These arguments form a bounding box in which the scroll bar will be drawn.

**style:** HOWL logically ORs this value with the Windows styles (`w.WS_CHILD` | `w.WS_VISIBLE`) when creating the scroll bar. At the very least, this field should contain `w.SBS_HORZ` for a horizontal scroll bar or `w.SBS_VERT` for a vertical scroll bar. Please see the Windows documentation or the HLA *w.hhf* header file for additional scroll bar style (`SBS_*`) constants.

**onChange:** this is the address (which can be NULL) of an `onChange` `widgetProc` procedure. The constructor will initialize the `onChange` field with this value.

`enable,`

`disable`

These two methods will enable or disable the scroll bar on the form. A disabled scroll bar is still visible, but the user will be unable to interact with it.

<code>show,</code>	
<code>hide</code>	These two methods will make a scroll bar visible or invisible on the form.
<code>get_position</code>	This function returns the current scroll bar position as a 32-bit value in the EAX register. Note that you can call this method from any of the event notification procedures except <code>onThumbTrack</code> to obtain the true 32-bit position of the thumb control (rather than limiting yourself to the upper 16 bits of the <code>wParam</code> parameter). If you call this method from the <code>onThumbTrack</code> notification procedure, it will not return the current position of the thumb, instead it will return the last position before the user started dragging the thumb around. Sadly, there is no way to obtain a 32-bit current thumb position while dragging the thumb.
<code>set_position</code>	This method sets the current thumb position on the scroll bar. The argument you pass to this method must be between the low and high range values for the scroll bar.
<code>get_lowRange</code>	This method returns (in EAX) the current lower range for the scroll bar. This is typically zero, but you can program any 32-bit value you desire (see <code>set_range</code> ).
<code>get_hiRange</code>	This method returns (in EAX) the current upper range for the scroll bar.
<code>set_range</code>	This method lets you set the low and high range values for the scroll bar. The default range is 0..100, but you can set any 32-bit values you like. Note that if you intend to use the <code>onThumbTrack</code> notification, you should limit the range to 16-bit values.  <b>low:</b> the lower bound of the scroll bar range <b>high:</b> the upper bound of the scroll bar range.
<code>get_onChange,</code>	
<code>get_onThumbPosn,</code>	
<code>get_onThumbTrack,</code>	
<code>get_onLineDown,</code>	
<code>get_onLineUp,</code>	
<code>get_onLineLeft,</code>	
<code>get_onLineRight,</code>	
<code>get_onEndScroll,</code>	
<code>get_onPageDown,</code>	
<code>get_onPageUp,</code>	
<code>get_onPageLeft,</code>	
<code>get_onPageRight,</code>	
<code>get_onTop,</code>	
<code>get_onBottom</code>	These are the accessor methods for the corresponding <code>widgetProc</code> pointer data fields in this class.
<code>get_lineInc,</code>	
<code>get_pageInc</code>	These are the accessor methods for the corresponding data fields in the class. Applications should always use these accessor methods to read the values of the <code>lineInc</code> and <code>pageInc</code> fields.
<code>set_onChange,</code>	
<code>set_onThumbPosn,</code>	
<code>set_onThumbTrack,</code>	
<code>set_onLineDown,</code>	
<code>set_onLineUp,</code>	
<code>set_onLineLeft,</code>	
<code>set_onLineRight,</code>	
<code>set_onEndScroll,</code>	

set_onPageDown,	
set_onPageUp,	
set_onPageLeft,	
set_onPageRight,	
set_onTop,	
set_onBottom	These are the mutator functions for all the widgetProc pointer fields in the class.
set_lineInc,	
set_pageInc	These are the mutator methods for the corresponding data fields in the class. Applications should always use these accessor methods to write the values of the <code>lineInc</code> and <code>pageInc</code> fields.
get_textColor,	
set_textColor,	
get_bkgColor,	
set_bkgColor	These accessor/mutator functions get and set the text and background colors that the widget uses.
processMessage	This is a private method in the class. Applications must not call this method.

### 18.3.17.2 wTrackBar\_t

`wTrackBar_t` objects are very similar in use to a scroll bar insofar as they provide a slider control that the user can move between two extremes on the widget. However, whereas scroll bars have a specific user interface purpose (moving the view through a window), track bars are generalized numeric input devices. Their purpose is to input a numeric value between a low range and a high range via a slider control.

Like scroll bars, you should limit the range of trackbar responses to 16 bits if you intend to use the `onThumbTrack` notification.

```

wTrackBar_t:
    class inherits( wVisual_t );

    var
        align( 4 );
        wTrackBar_private:
            record

                onChange           :widgetProc;
                onThumbPosn        :widgetProc;
                onThumbTrack       :widgetProc;
                onBottom           :widgetProc;
                onLineDown         :widgetProc;
                onLineUp           :widgetProc;
                onTop              :widgetProc;
                onEndtrack         :widgetProc;
                onPageDown         :widgetProc;
                onPageUp           :widgetProc;

            endrecord;

    procedure create_wTrackBar
    (
        wtbName      :string;
        parent       :dword;
        x            :dword;
        y            :dword;
        width        :dword;
    )

```

```

        height      :dword;
        style       :dword;
        onChange    :widgetProc
    );    external;

method get_position; @returns( "eax" );           external;
method set_position( position:dword );           external;

method get_lowRange; @returns( "eax" );           external;
method get_hiRange;  @returns( "eax" );           external;
method set_range( low:dword; high:dword );        external;

method get_onChange;      @returns( "eax" );       external;
method get_onThumbPosn;   @returns( "eax" );       external;
method get_onThumbTrack;  @returns( "eax" );       external;
method get_onBottom;      @returns( "eax" );       external;
method get_onLineDown;    @returns( "eax" );       external;
method get_onLineUp;      @returns( "eax" );       external;
method get_onTop;         @returns( "eax" );       external;
method get_onEndtrack;    @returns( "eax" );       external;
method get_onPageDown;    @returns( "eax" );       external;
method get_onPageUp;      @returns( "eax" );       external;

method set_onChange( onChange:widgetProc );       external;
method set_onThumbPosn( onThumbPosn:widgetProc ); external;
method set_onThumbTrack( onThumbTrack:widgetProc ); external;
method set_onBottom( onBottom:widgetProc );       external;
method set_onLineDown( onLineDown:widgetProc );   external;
method set_onLineUp( onLineUp:widgetProc );        external;
method set_onTop( onTop:widgetProc );              external;
method set_onEndtrack( onEndtrack:widgetProc );    external;
method set_onPageDown( onPageDown:widgetProc );    external;
method set_onPageUp( onPageUp:widgetProc );        external;

override method processMessage;                   external;

endclass;

```

onChange	This field, if non-NULL, points at a <code>widgetProc</code> procedure that HOWL will call whenever the user changes the position of the slider on the track bar (using any means to change the value). Upon entry into the <code>widgetProc</code> procedure, the H.O. word of the <code>wParam</code> argument contains the current track bar position.
onThumbPosn	This field, if non-NULL, points at a <code>widgetProc</code> procedure that HOWL will call whenever the user changes the position of the slider on the track bar by dragging the slider. HOWL calls this procedure after the user has released the mouse button while dragging the slider control. Upon entry into the <code>widgetProc</code> procedure, the H.O. word of the <code>wParam</code> argument contains the current scroll bar position.
onThumbTrack	This field, if non-NULL, points at a <code>widgetProc</code> procedure that HOWL will call while the user is dragging the slider around the track bar. An application can use this notification to dynamically adjust the system during slider movement. Upon entry into the <code>widgetProc</code> procedure, the H.O. word of the <code>wParam</code> argument contains the current track bar slider position.
onBottom	This field, if non-NULL, points at a <code>widgetProc</code> procedure that HOWL will call whenever the user moves the slider all the way to the bottom on a vertical track bar or all the way to the left on a horizontal track bar. Upon entry into the <code>widgetProc</code> procedure, the H.O. word of the <code>wParam</code> argument contains the current track bar position (which should be the maximum value).

<code>onLineDown</code>	This field, if non-NULL, points at a <code>widgetProc</code> procedure that HOWL will call whenever the user changes the position of the slider on a track bar by pressing the down arrow or the right arrow key on the keyboard. Upon entry into the <code>widgetProc</code> procedure, the H.O. word of the <code>wParam</code> argument contains the current track bar position.
<code>onLineUp</code>	This field, if non-NULL, points at a <code>widgetProc</code> procedure that HOWL will call whenever the user changes the position of the slider on a track bar by pressing the up arrow or left arrow key on the keyboard. Upon entry into the <code>widgetProc</code> procedure, the H.O. word of the <code>wParam</code> argument contains the current track bar position.
<code>onTop</code>	This field, if non-NULL, points at a <code>widgetProc</code> procedure that HOWL will call whenever the user moves the slider all the way to the top on a vertical track bar or all the way to the left on a horizontal track bar. Upon entry into the <code>widgetProc</code> procedure, the H.O. word of the <code>wParam</code> argument contains the current track bar position (which should be the maximum value).
<code>onEndTrack</code>	This field, if non-NULL, points at a <code>widgetProc</code> procedure that HOWL will call after any trackbar operation is complete. Upon entry into the <code>widgetProc</code> procedure, the H.O. word of the <code>wParam</code> argument contains the current track bar position.
<code>onPageDown</code>	This field, if non-NULL, points at a <code>widgetProc</code> procedure that HOWL will call whenever the user changes the position of the slider on a vertical scroll bar by clicking on the track bar between the slider and the left or bottom end of the track bar. Upon entry into the <code>widgetProc</code> procedure, the H.O. word of the <code>wParam</code> argument contains the current track bar position.
<code>onPageUp</code>	This field, if non-NULL, points at a <code>widgetProc</code> procedure that HOWL will call whenever the user changes the position of the slider on a track bar by clicking on the track bar between the slider and the top or right side of the track bar. Upon entry into the <code>widgetProc</code> procedure, the H.O. word of the <code>wParam</code> argument contains the current track bar position.
<code>create_wTrackBar</code>	<p>This is the constructor for the <code>wTrackBar_t</code> class. If you call this as a class procedure (e.g., "<code>wTrackBar_t.create_wTrackBar</code>") then this procedure will allocate storage for a new <code>wTrackBar_t</code> object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then <code>create_wTrackBar</code> will initialize that object in-place.</p> <p><b>wtbName:</b> HOWL assigns this string to the <code>_name</code> data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.</p> <p><b>parent:</b> this is the handle of the <code>wView_t</code> or <code>wForm_t</code> object on which the track bar will be drawn.</p> <p><b>x, y, width, height:</b> These arguments form a bounding box in which the track bar will be drawn.</p> <p><b>style:</b> HOWL logically ORs this value with the Windows styles (<code>w.WS_CHILD</code>   <code>w.WS_VISIBLE</code>   <code>w.TBS_AUTOTICKS</code>) when creating the scroll bar. At the very least, this field should contain <code>w.TBS_HORZ</code> for a horizontal track bar or <code>w.TBS_VERT</code> for a vertical track bar. Please see the Windows documentation or the HLA <i>w.hhf</i> header file for additional track bar style (<code>TBS_*</code>) constants.</p> <p><b>onChange:</b> this is the address (which can be NULL) of an <code>onChange</code> <code>widgetProc</code> procedure. The constructor will initialize the <code>onChange</code> field with this value.</p>
<code>get_position</code>	This function returns the current track bar position as a 32-bit value in the EAX register. Note that you can call this method from any of the event notification procedures except <code>onThumbTrack</code> to obtain the true 32-bit position of the slider control (rather than limiting yourself to the upper 16 bits of the <code>wParam</code> parameter). If you call this method from the <code>onThumbTrack</code> notification procedure, it will not return the current position of the slider,

instead it will return the last position before the user started dragging the slider around. Sadly, there is no way to obtain a 32-bit current thumb position while dragging the thumb.

`set_position` This method sets the current thumb position on the track bar. The argument you pass to this method must be between the low and high range values for the track bar.

`get_lowRange` This method returns (in EAX) the current lower range for the track bar. This is typically zero, but you can program any 32-bit value you desire (see `set_range`).

`get_hiRange` This method returns (in EAX) the current upper range for the track bar.

`set_range` This method lets you set the low and high range values for the track bar. The default range is 0..100, but you can set any 32-bit values you like. Note that if you intend to use the `onThumbTrack` notification, you should limit the range to 16-bit values.

**low:** the lower bound of the track bar range

**high:** the upper bound of the track bar range.

`get_onChange,`  
`get_onThumbPosn,`  
`get_onThumbTrack,`  
`get_onLineDown,`  
`get_onLineUp,`  
`get_onLineLeft,`  
`get_onLineRight,`  
`get_onEndtrack,`  
`get_onPageDown,`  
`get_onPageUp,`  
`get_onPageLeft,`  
`get_onPageRight,`  
`get_onTop,`  
`get_onBottom`

These are the accessor methods for the corresponding `widgetProc` pointer data fields in this class.

`set_onChange,`  
`set_onThumbPosn,`  
`set_onThumbTrack,`  
`set_onLineDown,`  
`set_onLineUp,`  
`set_onLineLeft,`  
`set_onLineRight,`  
`set_onEndtrack,`  
`set_onPageDown,`  
`set_onPageUp,`  
`set_onPageLeft,`  
`set_onPageRight,`  
`set_onTop,`  
`set_onBottom`  
`processMessage`

These are the mutator functions for all the `widgetProc` pointer fields in the class.

This is a private method in the class. Applications must not call this method.

### 18.3.18 Up/Down Arrows

An up/down arrow control is a pair of (stacked) arrow buttons that that user can click on in order to increment or decrement a value. Up/down arrow widgets can be stand-alone or they can be attached to a "buddy" edit box widget (`wUpDownEditBox_t` widgets). When attached to a buddy edit box, the up/down arrow control reflects the current value of the control in the edit box.

#### 18.3.18.1 `wUpDown_t`

The `wUpDown_t` class is used to create up/down arrow objects on a form.

```
wUpDown_t:
    class inherits( wClickable_t );

    procedure create_wUpDown
    (
        wudName      :string;
        parent       :dword;
        alignment     :dword;
        x             :dword;
        y             :dword;
        width         :dword;
        height        :dword;
        lowerRange    :dword;
        upperRange    :dword;
        initialPosn   :dword;
        onClick       :widgetProc
    );    external;

    method get_lowerRange;    @returns( "eax" );    external;
    method get_upperRange;    @returns( "eax" );    external;
    method get_position;      @returns( "eax" );    external;

    method set_lowerRange( lowerRange:word );    external;
    method set_upperRange( upperRange:word );    external;
    method set_position(   position  :word );    external;

    override method processMessage;    external;

endclass;

create_wUpDown
```

This is the constructor for the `wUpDown_t` class. If you call this as a class procedure (e.g., "`wUpDown_t.create_wUpDown`") then this procedure will allocate storage for a new `wUpDown_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wUpDown` will initialize that object in-place.

**wudName:** HOWL assigns this string to the `_name` data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

**parent:** this is the handle of the `wView_t` or `wForm_t` object on which the track bar will be drawn.

**alignment:** this is one of the following Up/Down window styles:

`UDS_ALIGNLEFT`

Positions the up-down control next to the left edge of the buddy window. The buddy window is moved to the right and its width decreased to accommodate the width of the up-down control.

`UDS_ALIGNRIGHT`

Positions the up-down control next to the right edge of the buddy window. The width of the buddy

	window is decreased to accommodate the width of the up-down control.
UDS_ARROWKEYS	Causes the up-down control to increment and decrement the position when the UP ARROW and DOWN ARROW keys are pressed.
UDS_AUTOBUDDY	Automatically selects the previous window in the Z order as the up-down control's buddy window.
UDS_HORZ	Causes the up-down control's arrows to point left and right instead of up and down.
UDS_NOTHOUSANDS	Does not insert a thousands separator between every three decimal digits.
UDS_SETBUDDYINT	Causes the up-down control to set the text of the buddy window (using the WM_SETTEXT message) when the position changes. The text consists of the position formatted as a decimal or hexadecimal string.
UDS_WRAP	Causes the position to "wrap" if it is incremented or decremented beyond the ending or beginning of the range.

**x, y, width, height:** These arguments form a bounding box in which the up/down arrow will be drawn. If `buddy` contains a non-NULL value, then Windows will ignore these values and will, instead, attach the arrows to the buddy edit box.

**lowerRange:** This is the minimum value that the up/down arrow control will decrement to. This value is typically zero, but it can be any 16-bit value that is less than the `upperRange` value. If the up/down control's current value is equal to `lowerRange` and the user presses on the down arrow, Windows will ignore the decrement request.

**upperRange:** This is the maximum value that the up/down arrow control will increment to. This value can be any 16-bit value that is greater than the `lowerRange` value. If the up/down control's current value is equal to `upperRange` and the user presses on the up arrow, Windows will ignore the increment request.

**initialPosn:** This is the initial value of the up/down arrow control. It must be a 16-bit value in the range `lowerRange..upperRange`.

**onClick:** this is the address (which can be NULL) of an `onClick` widgetProc procedure. The constructor will initialize the `onClick` field with this value. (Note that the `onClick` field is inherited from the `wClickable_t` parent class).

<code>get_lowerRange</code>	This method retrieves the 16-bit lower range value for the up/down arrow control. It returns this 16-bit value (zero extended) in the EAX register.
<code>get_upperRange</code>	This method retrieves the 16-bit upper range value for the up/down arrow control. It returns this 16-bit value (zero extended) in the EAX register.
<code>get_position</code>	This method retrieves the 16-bit position value for the up/down arrow control. It returns this 16-bit value (zero extended) in the EAX register.
<code>set_lowerRange</code>	This method sets the 16-bit lower range value for the up/down arrow control. Applications should limit the parameter to a 16-bit value.
<code>set_upperRange</code>	This method sets the 16-bit upper range value for the up/down arrow control. Applications should limit the parameter to a 16-bit value.
<code>set_position</code>	This method sets the 16-bit position value for the up/down arrow control. Applications should limit the parameter to a 16-bit value.
<code>processMessage</code>	This is a private method. Applications must not call this method.



### 18.3.18.2 wUpDownEditBox\_t

The wUpDownEditBox\_t class is used to create a combination edit box and up/down arrow on a form.

```

wUpDownEditBox_t:
  class inherits( wabsEditBox_t );
  var
    wUpDownEditBox_private:
      record

        lowerRange      :dword;
        upperRange      :dword;
        upDownHandle    :dword;
        upDownStyle     :dword;
        onUpDown        :widgetProc;

      endrecord;

  procedure create_wUpDownEditBox
  (
    wudName      :string;
    initialTxt   :string;
    parent       :dword;
    style        :dword;
    upDownStyle  :dword;
    x            :dword;
    y            :dword;
    width        :dword;
    height       :dword;
    lowerRange   :dword;
    upperRange   :dword;
    initialPosn  :dword;
    onChange     :widgetProc;
    onUpDown     :widgetProc
  );  external;

  method get_lowerRange; @returns( "eax" );      external;
  method get_upperRange; @returns( "eax" );      external;
  method get_position;   @returns( "eax" );      external;

  method set_lowerRange( lowerRange:word );      external;
  method set_upperRange( upperRange:word );      external;
  method set_position(   position  :word );      external;

  override method show;                          external;
  override method hide;                          external;
  override method enable;                        external;
  override method disable;                      external;

  override method processMessage;                external;

endclass;

create_wUpDownEditBox

```

This is the constructor for the wUpDownEditBox\_t class. If you call this as a class procedure (e.g., "wUpDownEditBox\_t.create\_wUpDown") then this procedure will allocate storage for a new wUpDownEditBox\_t object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then create\_wUpDown will initialize that object in-place.

**wudName:** HOWL assigns this string to the `_name` data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

**parent:** this is the handle of the `wView_t` or `wForm_t` object on which the track bar will be drawn.

**alignment:** this is one of the following Up/Down window styles:

<code>UDS_ALIGNLEFT</code>	Positions the up-down control next to the left edge of the buddy window. The buddy window is moved to the right and its width decreased to accommodate the width of the up-down control.
<code>UDS_ALIGNRIGHT</code>	Positions the up-down control next to the right edge of the buddy window. The width of the buddy window is decreased to accommodate the width of the up-down control.
<code>UDS_ARROWKEYS</code>	Causes the up-down control to increment and decrement the position when the UP ARROW and DOWN ARROW keys are pressed.
<code>UDS_AUTOBUDDY</code>	Automatically selects the previous window in the Z order as the up-down control's buddy window.
<code>UDS_HORZ</code>	Causes the up-down control's arrows to point left and right instead of up and down.
<code>UDS_NOTHOUSANDS</code>	Does not insert a thousands separator between every three decimal digits.
<code>UDS_SETBUDDYINT</code>	Causes the up-down control to set the text of the buddy window (using the <code>WM_SETTEXT</code> message) when the position changes. The text consists of the position formatted as a decimal or hexadecimal string.
<code>UDS_WRAP</code>	Causes the position to "wrap" if it is incremented or decremented beyond the ending or beginning of the range.

**x, y, width, height:** These arguments form a bounding box in which the up/down arrow will be drawn. If `buddy` contains a non-NULL value, then Windows will ignore these values and will, instead, attach the arrows to the buddy edit box.

**lowerRange:** This is the minimum value that the up/down arrow control will decrement to. This value is typically zero, but it can be any 16-bit value that is less than the `upperRange` value. If the up/down control's current value is equal to `lowerRange` and the user presses on the down arrow, Windows will ignore the decrement request.

**upperRange:** This is the maximum value that the up/down arrow control will increment to. This value can be any 16-bit value that is greater than the `lowerRange` value. If the up/down control's current value is equal to `upperRange` and the user presses on the up arrow, Windows will ignore the increment request.

**initialPosn:** This is the initial value of the up/down arrow control. It must be a 16-bit value in the range `lowerRange..upperRange`.

**onClick:** this is the address (which can be NULL) of an `onClick` widgetProc procedure. The constructor will initialize the `onClick` field with this value. (Note that the `onClick` field is inherited from the `wClickable_t` parent class).

<code>get_lowerRange</code>	This method retrieves the 16-bit lower range value for the up/down arrow control. It returns this 16-bit value (zero extended) in the EAX register.
<code>get_upperRange</code>	This method retrieves the 16-bit upper range value for the up/down arrow control. It returns this 16-bit value (zero extended) in the EAX register.

<code>get_position</code>	This method retrieves the 16-bit position value for the up/down arrow control. It returns this 16-bit value (zero extended) in the EAX register.
<code>set_lowerRange</code>	This method sets the 16-bit lower range value for the up/down arrow control. Applications should limit the parameter to a 16-bit value.
<code>set_upperRange</code>	This method sets the 16-bit upper range value for the up/down arrow control. Applications should limit the parameter to a 16-bit value.
<code>set_position</code>	This method sets the 16-bit position value for the up/down arrow control. Applications should limit the parameter to a 16-bit value.
<code>processMessage</code>	This is a private method. Applications must not call this method.

## 18.3.19 Icons

Icons are generally 16x16 or 32x32 bitmapped objects drawn on the screen.

### 18.3.19.1 `wIcon_t`

A `wIcon_t` object on a form can draw a user-defined icon or a system icon.

```

wIcon_t:
  class inherits( wSurface_t );

  var
    align( 4 );
    wIcon_private:
      record

        iconName      :string;
        iconHandle     :dword;

      endrecord;

  procedure create_wIcon
  (
    wiName      :string;
    iconName    :string;
    parent      :dword;
    x           :dword;
    y           :dword;
    width       :dword;
    height      :dword;
    bkgColor    :dword
  ); external;

  method get_iconName;    @returns( "eax" );    external;
  method load_icon( iconName:string );          external;
  override method destroy;                      external;
  override method processMessage;                external;

endclass;

```

<code>iconName</code>	This is either the name of an icon resource within the executable file, or a special numeric value (less than \$1_0000) that specifies a system icon. Note that this field does not contain the name of an icon file on the disk. Applications should not set the value of this field directly. Instead, they should use the constructor or the <code>load_icon</code> method to set the icon name.
<code>iconHandle</code>	This is a handle for the icon resource associated with this <code>wIcon_t</code> object. This is a private field and applications should not read or write its value.

<code>create_wIcon</code>	<p>This is the constructor for the <code>wIcon_t</code> class. If you call this as a class procedure (e.g., "<code>wIcon_t.create_wIcon</code>") then this procedure will allocate storage for a new <code>wIcon_t</code> object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then <code>wIcon</code> will initialize that object in-place.</p> <p><b>wiName:</b> HOWL assigns this string to the <code>_name</code> data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.</p> <p><b>iconName:</b> this is the name of the icon resource in the executable file to associate with the icon (if this value is a string), or it is a value less than <code>\$1_0000</code> that specifies a system icon value. The following are valid non-string values to supply to this parameter: <code>w.IDI_APPLICATION</code>, <code>w.IDI_ASTERISK</code>, <code>w.IDI_EXCLAMATION</code>, <code>w.IDI_HAND</code>, <code>w.IDI_QUESTION</code>, and <code>w.IDI_WINLOGO</code>.</p> <p><b>parent:</b> this is the handle of the <code>wView_t</code> or <code>wForm_t</code> object on which the icon will be drawn.</p> <p><b>x, y, width, height:</b> These arguments form a bounding box in which the up/down arrow will be drawn. If <code>buddy</code> contains a non-NULL value, then Windows will ignore these values and will, instead, attach the arrows to the buddy edit box.</p> <p><b>bkgColor:</b> this is the RGB background color that HOWL will fill the bounding rectangle with if the bounding rectangle is larger than the icon.</p>
<code>load_icon</code>	This method loads the icon resource whose name you specify as the string parameter into the icon object. The parameter must either be the (string) name of an icon resource in the executable file or one of the system-defined icon constants (see the discussion for the constructor).
<code>destroy</code>	This is the destructor for the <code>wIcon_t</code> object. Applications won't normally call this destructor for icons attached to some form (or other container) as destroying that container will automatically invoke the destructor for the icon. However, if you create a stand-alone icon object and don't attach it to some container object, then you should call the icon's destructor when you are done with the icon to free the associated system resources.
<code>processMessage</code>	This is a private method. Applications must not call it.

## 18.3.20 Text

HOWL provides two classes for dealing with text: the `wFont_t` class and the `wLabel_t` class. Unlike most concrete classes in HOWL, `wFont_t` objects are not widgets (that is, controls that appear visually on a form). Font objects are associated with `wLabel_t` objects (which do appear on a form) and other text, but are not visual items themselves.

### 18.3.20.1 wFont\_t

Font objects represent a particular typeface for use by `wLabel_t` and other text objects in HOWL. Note that `wFont_t` objects are somewhat unique amongst the HOWL concrete classes insofar as there is no statement in the HOWL declarative language to create a font object. In general, an application will create all the fonts it needs in the `appStart` procedure and it will call the destructors for those font objects in the `appTerminate` procedure.

```

wFont_t:
  class inherits( wBase_t );

  var
    align( 4 );
    wFont_private:
      record

        family           :byte;
```

```

        bold           :boolean;
        italic         :boolean;
        underline      :boolean;
        strikeouts      :boolean;
        monospaced     :boolean;
        align( 4 );

        faceName       :string;
        size            :uns32;

    endrecord;

procedure create_wFont
(
    wfName             :string;
    parentHandle       :dword;
    faceName           :string;
    family             :byte;
    size               :uns32;
    bold               :boolean;
    italic              :boolean;
    underline          :boolean;
    strikeouts          :boolean;
    monospaced         :boolean
); external;

override method destroy;                external;

// Accessor functions:

method get_facename;    @returns( "eax" ); external;
method get_size;        @returns( "eax" ); external;
method get_family;      @returns( "al" ); external;
method get_bold;        @returns( "al" ); external;
method get_italic;      @returns( "al" ); external;
method get_underline;   @returns( "al" ); external;
method get_strikeout;   @returns( "al" ); external;
method get_monospaced;  @returns( "al" ); external;

endclass;

```

faceName	This string is the name of the Windows typeface to use for the font. This is a string such as "Courier New" or "Times New Roman". If this field is NULL or is the empty string, then Windows will pick an appropriate font that matches the other font characteristics. If Windows cannot find the specified font name, it will find the closest one that matches the font characteristics you provide. Applications should not write values to this field.
size	This is the size, in points, for the typeface. Applications should not write values to this field.
family	<p>This is one of the following constants:</p> <p>w.FF_DECORATIVE Novelty fonts. Old English is an example.</p> <p>w.FF_DONTCARE Don't care or don't know.</p> <p>w.FF_MODERN Fonts with constant stroke width, with or without serifs. Pica, Elite, and Courier New® are examples.</p> <p>w.FF_ROMAN Fonts with variable stroke width and with serifs. MS® Serif is an example.</p>

	w.FF_SCRIPT	Fonts designed to look like handwriting. Script and Cursive are examples.
	w.FF_SWISS	Fonts with variable stroke width and without serifs. MS Sans Serif is an example.
	Applications should not write values to the <code>family</code> field.	
<code>bold</code>	True for boldfaced fonts, false for normal weight fonts. Applications should not write values to this field.	
<code>italic</code>	True for italic slant fonts, false for normal fonts. Applications should not write values to this field.	
<code>underline</code>	True for underlined fonts, false for normal fonts. Applications should not write values to this field.	
<code>strikeout</code>	True for strikeout fonts, false for normal fonts. Applications should not write values to this field.	
<code>monospaced</code>	True for monospaced fonts, false for proportional fonts. Applications should not write values to this field.	
<code>create_wFont</code>	<p>This is the constructor for the <code>wFont_t</code> class. If you call this as a class procedure (e.g., "<code>wFont_t.create_wFont</code>") then this procedure will allocate storage for a new <code>wFont_t</code> object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then <code>create_wFont</code> will initialize that object in-place.</p> <p><b>wfName:</b> HOWL assigns this string to the <code>_name</code> data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.</p> <p>The remaining parameters all correspond to the data fields for this class. See their descriptions for more details. Calling the constructor is the only legal way to write values to the <code>wFont_t</code> class' data fields.</p>	
<code>destroy</code>	<p>This is the destructor for the <code>wFont_t</code> class. Because fonts aren't normally attached to a form (and you cannot create them via the HOWL declarative language), it is usually the applications responsibility to call the destructor for a font when the application is done using it. Alternately, an application can insert a font into a container object and let that container destroy the font object when the container is destroyed.</p>	

### 18.3.20.2 wLabel\_t

A `wLabel_t` object displays static text on a form. Actually, "static" is a bit of a misnomer because an application can change the text with a method call, but Windows, HOW, and the user do not change this text behind the application's back.

```

wLabel_t:
  class inherits( wVisual_t );
  var
    align( 4 );
    wLabel_private:
      record

        caption      :string;
        font          :wFont_p;
        alignment     :dword;
        foreColor     :dword;

      endrecord;

  procedure create_wLabel
  (
    wName            :string;

```

```

        caption      :string;
        parent       :dword;
        x            :dword;
        y            :dword;
        width        :dword;
        height       :dword;
        alignment     :dword;
        foreColor    :dword;
        bkgColor     :dword
    );  external;

    override method destroy;                                external;
    override method processMessage;                          external;

    method get_font;          @returns( "eax" );             external;
    method get_caption;       @returns( "eax" );             external;
    method a_get_caption;     @returns( "eax" );             external;
    method get_foreColor;     @returns( "eax" );             external;

    method set_font( font:wFont_p );                          external;
    method set_caption( caption:string );                      external;
    method set_foreColor( foreColor:dword );                  external;

endclass;

```

caption	This data field points at the textual data that the wLabel_t object will display on a form. Applications should never read or write this data field directly, they should always use the get_caption and set_caption accessor/mutator methods to manipulate this string.																		
font	This is a pointer to a wFont_t object or the value NULL. If this field contains NULL, Windows will pick an appropriate system font and use that to draw the label's text. If this field contains a pointer to a wFont_t object, this Windows will use that particular font to draw the label's text. Applications should never read or write this data field directly, they should always use the get_font and set_font accessor/mutator methods to manipulate this value.																		
alignment	Applications should not write to this field. It's value is set by the constructor. This field should contain one of the following constants: <table> <tr> <td>w.DT_BOTTOM</td><td>Bottom-justifies text. This value must be combined with DT_SINGLELINE.</td></tr> <tr> <td>w.DT_CENTER</td><td>Centers text horizontally.</td></tr> <tr> <td>w.DT_EXPANDTABS</td><td>Expands tab characters. The default number of characters per tab is eight.</td></tr> <tr> <td>w.DT_LEFT</td><td>Aligns text to the left.</td></tr> <tr> <td>w.DT_NOPREFIX</td><td>Turns off processing of prefix characters. Normally, DrawText interprets the mnemonic-prefix character &amp; as a directive to underscore the character that follows, and the mnemonic-prefix characters &amp;&amp; as a directive to print a single &amp;. By specifying DT_NOPREFIX, this processing is turned off.</td></tr> <tr> <td>w.DT_RIGHT</td><td>Aligns text to the right.</td></tr> <tr> <td>w.DT_SINGLELINE</td><td>Displays text on a single line only. Carriage returns and linefeeds do not break the line.</td></tr> <tr> <td>w.DT_TOP</td><td>Top-justifies text (single line only).</td></tr> <tr> <td>w.DT_VCENTER</td><td>Centers text vertically (single line only).</td></tr> </table>	w.DT_BOTTOM	Bottom-justifies text. This value must be combined with DT_SINGLELINE.	w.DT_CENTER	Centers text horizontally.	w.DT_EXPANDTABS	Expands tab characters. The default number of characters per tab is eight.	w.DT_LEFT	Aligns text to the left.	w.DT_NOPREFIX	Turns off processing of prefix characters. Normally, DrawText interprets the mnemonic-prefix character & as a directive to underscore the character that follows, and the mnemonic-prefix characters && as a directive to print a single &. By specifying DT_NOPREFIX, this processing is turned off.	w.DT_RIGHT	Aligns text to the right.	w.DT_SINGLELINE	Displays text on a single line only. Carriage returns and linefeeds do not break the line.	w.DT_TOP	Top-justifies text (single line only).	w.DT_VCENTER	Centers text vertically (single line only).
w.DT_BOTTOM	Bottom-justifies text. This value must be combined with DT_SINGLELINE.																		
w.DT_CENTER	Centers text horizontally.																		
w.DT_EXPANDTABS	Expands tab characters. The default number of characters per tab is eight.																		
w.DT_LEFT	Aligns text to the left.																		
w.DT_NOPREFIX	Turns off processing of prefix characters. Normally, DrawText interprets the mnemonic-prefix character & as a directive to underscore the character that follows, and the mnemonic-prefix characters && as a directive to print a single &. By specifying DT_NOPREFIX, this processing is turned off.																		
w.DT_RIGHT	Aligns text to the right.																		
w.DT_SINGLELINE	Displays text on a single line only. Carriage returns and linefeeds do not break the line.																		
w.DT_TOP	Top-justifies text (single line only).																		
w.DT_VCENTER	Centers text vertically (single line only).																		

	<code>w.DT_WORDBREAK</code>	Breaks words. Lines are automatically broken between words if a word would extend past the edge of the rectangle specified by the <code>lpRect</code> parameter. A carriage return-linefeed sequence also breaks the line.
<code>foreColor</code>		This is the foreground color for the text (the RGB color used to draw the actual characters). Applications should never read or write this data field directly, they should always use the <code>get_foreColor</code> and <code>set_foreColor</code> accessor/mutator methods to manipulate this value.
<code>create_wLabel</code>		<p>This is the constructor for the <code>wLabel_t</code> class. If you call this as a class procedure (e.g., "<code>wLabel_t.create_wLabel</code>") then this procedure will allocate storage for a new <code>wLabel_t</code> object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then <code>create_wLabel</code> will initialize that object in-place.</p> <p><b>wName:</b> HOWL assigns this string to the <code>_name</code> data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.</p> <p><b>caption:</b> this is string data HOWL will associate with the label object. Note that HOWL will make an internal copy (on the heap) of this string.</p> <p><b>parent:</b> this is the handle of the <code>wView_t</code> or <code>wForm_t</code> object on which the label will be drawn.</p> <p><b>x, y, width, height:</b> These arguments form a bounding box in which the up/down arrow will be drawn. If <code>buddy</code> contains a non-NULL value, then Windows will ignore these values and will, instead, attach the arrows to the buddy edit box.</p> <p><b>alignment:</b> this is one of the alignment constants described under the alignment field, earlier. If this field is zero, <code>w.DT_LEFT</code> is the default value.</p> <p><b>foreColor:</b> this is an RGB value that specifies the foreground color for the label.</p> <p><b>bkgColor:</b> this is an RGB value that specifies the background color for the label.</p>
<code>destroy</code>		This is the destructor for the label class. Normally, the container form holding the label will call this destructor automatically when you destroy the form. If you create a label manually and don't insert it into the widget list of a container object, then you will need to call this destructor yourself.
<code>processMessage</code>		This is a private method. Applications should not call this method.
<code>get_font</code>		This is the accessor method that returns a pointer to the <code>wFont_t</code> object (or NULL) pointed at by the <code>font</code> data field.
<code>get_caption</code>		This returns the caption string pointer in EAX for read-only access. The application must not modify this string or HOWL will display inconsistent results.
<code>a_get_caption</code>		This method returns a pointer to a copy of the <code>caption</code> string (that it allocates on the heap) in the EAX register. It is the caller's responsibility to free the storage associated with this string when it is done using the string data.
<code>get_foreColor</code>		This accessor method returns the current foreground color that the <code>wLabel_t</code> object uses to draw the text.
<code>set_font</code>		This method is the mutator function for the <code>font</code> data field. Applications should always call this method to change the font of a <code>wLabel_t</code> object.
<code>set_caption</code>		This method is the mutator function for the <code>caption</code> data field. Applications should always call this method to change the string associated with a <code>wLabel_t</code> object.
<code>set_foreColor</code>		This method is the mutator function for the <code>foreColor</code> data field. Applications should always call this method to change the foreground color associated with a <code>wLabel_t</code> object.



## 18.3.21 Views, Windows, and Tab Pages

A view is a container object that is also a surface upon which you can draw or place other widgets. In this sense, a view is very similar to a `wForm_t` object. Actually, a view is equivalent to the client area of a `wForm_t` object; that is, it's the application window without the title bar or system menu components.

### 18.3.21.1 `wTabPage_t`

The `wTabPage_t` class is basically a concrete implementation of the abstract `window_t` class.

```
wTabPage_t:
    class inherits( window_t );

    var
        align( 4 );
        wTabPage_private:
            record

                pageHandler :widgetProc;

            endrecord;

    procedure create_wTabPage
    (
        wpName           :string;
        parentWindowHandle :dword;
        handler          :widgetProc;
        x                :dword;
        y                :dword;
        width            :dword;
        height           :dword;
        fillColor        :dword
    ); external;

    override method processMessage;          external;

endclass;
```

**pageHandler** This is a `widgetProc` pointer to a user-defined `processMessage` function for the `wTabPage_t` object. If this pointer contains NULL, then HOWL will invoke the parent (`window_t`) `processMessage` handler. If all a `wForm_t` object is doing is containing other widgets, this is all that is necessary. However, if you're going to draw on the `wTabPage_t` object, then you will have to write your own `processMessage` function and intercept `w.WM_PAINT` and other messages. A discussion of the actual `pageHandler` procedure appears a little later in this section.

**create\_wTabPage** This is the constructor for the `wTabPage_t` class. If you call this as a class procedure (e.g., "`wTabPage_t.create_wTabPage`") then this procedure will allocate storage for a new `wTabPage_t` object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then `create_wTabPage` will initialize that object in-place.

**wpName:** HOWL assigns this string to the `_name` data field of the object. This string's value should be constant over the execution lifetime of the newly initialized object.

**parentWindowHandle:** this is the handle of the `wTabPage_t` or `wForm_t` object on which the `wTabPage_t` object will be placed.

**x, y, width, height:** These arguments form a bounding box in which the `wTabPage_t` object will be drawn.

**fillColor:** this is an RGB value that specifies the background color for the window.

**processMessage** This is a private method. Applications must not call it.

The `pageHandler` widgetProc needs some additional explanation. To begin with, although this pointer type is `widgetProc`, technically it ought to have the same prototype as the `processMessage` method (`hwnd`, `uMsg`, `wParam`, and `lParam` parameters). To overcome the difference in prototypes, HOWL passes to the `pageHandler` procedure the `uMsg` parameter value in the EAX register. The `hwnd` (window handle) argument is easily accessible as the `handle` field of the `wTabPage_t` object (accessible via the `thisPtr` parameter).

The `pageHandler` procedure is basically the Windows' `wndproc` procedure that handles the messages for the `wTabPage_t` object (that is, it's equivalent to a `wTabPage_t.processMessage` function). If you want to draw on the `wTabPage_t` surface, you'll need to intercept Windows' messages such as `w.WM_PAINT` and so on. Most of the time, you'll only want to handle a few of the messages sent to a `wTabPage_t` object yourself and you'll want to pass the rest of them on to a default message handler. In a pure Win32 application, you'd accomplish this by calling the `w.DefWindowProc` procedure. In an object-oriented system such as HOWL, however, what you really want to do is call the parent class' message handler (that is, `window_t.processMessage` in the case of a `wTabPage_t` object). Unfortunately, it's difficult to do this directly, so HOWL cheats and defines a special class procedure in the `window_t` class that you can call directly that is an alias for the `window_t.processMessage` method. This class procedure is called `window_t.view_t_processMessage` (so there is no mistaking its purpose). If ESI contains a pointer to the `wTabPage_t` object (and it must when you call `window_t.view_t_processMessage`), then you can call this procedure with the following statement:

```
(type window_t [esi]).view_t_processMessage
```

Here is a sample `pageHandler` procedure that does exactly the same thing as would happen if the `pageHandler` pointer were NULL (which is to call the `window_t.processMessage` method):

```
simplePH:widgetProc;
begin simplePH;

    mov( thisPtr, esi ); // Technically not needed, ESI contains this already
    (type window_t [esi]).view_t_processMessage // Call special alias proc
    (
        (type wTabPage_t [esi]).handle, // Handle to the wTabPage_t window
        eax,                          // Message code passed to us in EAX
        wParam,                       // Pass on the wParam value
        lParam                        // Pass on the lParam value
    );

end simplePH;
```

Of course, it doesn't make much sense to do nothing more than call the `view_t_processMessage` procedure. You could leave the `pageHandler` NULL and the `wTabPage_t` class would automatically do this for you (more efficiently, too). In general, you'll check for a few messages you need to handle and call `view_t_processMessage` as the default handler, e.g.:

```
typicalPH:widgetProc;
begin typicalPH;

    if( eax = someMessage ) then

        // code to handle someMessage

    elseif( eax = someOtherMessage ) then

        // code to handle someOtherMessage

    else // Default case

        mov( thisPtr, esi ); // Technically not needed, ESI contains this already
        (type window_t [esi]).view_t_processMessage // Call special alias proc
```

```

    (
        (type wTabPage_t [esi]).handle, // Handle to the wTabPage_t window
        eax,                          // Message code passed to us in EAX
        wParam,                       // Pass on the wParam value
        lParam,                       // Pass on the lParam value
    );

endif;

end typicalPH;

```

Please see the appropriate Windows documentation for details on how to handle message in a wndproc (message handling) procedure.

### 18.3.21.2 wView\_t

wView\_t objects are a concrete implementation of the wSurface\_t class. You use wView\_t objects to create borderless windows on which to draw things. Note that wView\_t is not a container class (like window\_t), so you cannot place other objects on a wView\_t window. Please see the discussion of the wSurface\_t class for more details concerning the capabilities of a wView\_t object.

```

wView_t:
    class inherits( wSurface_t );

    procedure create_wView
    (
        wsName      :string;
        exStyle     :dword;
        style       :dword;
        parent      :dword;
        x           :dword;
        y           :dword;
        width       :dword;
        height      :dword;
        bkgColor    :dword;
        visible     :boolean;
    ); external;

endclass;

```

### 18.3.21.3 window\_t

The window\_t type is functionally equivalent to wSurface\_t with one major difference: the window\_t class is derived from wContainer\_t rather than directly from wVisual\_t, so window\_t objects can contain other widgets.

```

window_t:
    class inherits( wContainer_t );

    var
        align( 4 );
        window_private:
            record

                // onPaint event pointer:

                onPaint      :widgetProc;

            endrecord;

```

```

procedure create_window
(
    wwName      :string;
    caption     :string;
    exStyle     :dword;
    style       :dword;
    parent      :dword;
    x           :dword;
    y           :dword;
    width       :dword;
    height      :dword;
    bkgColor    :dword;
    visible     :boolean
); external;

override method destroy;                external;
override method processMessage;         external;
override method onClose;               external;
override method onCreate;              external;

method get_onPaint;    @returns( "eax" );    external;
method set_onPaint( onPaint:widgetProc );    external;

procedure view_t_processMessage
(
    hwnd      :dword;
    uMsg      :dword;
    wParam    :dword;
    lParam    :dword
); external( "window_t_processMessage" );

endclass;

```

**onPaint** This widgetProc pointer is either NULL, or it points at a widgetProc procedure that HOWL will call whenever Windows sends the form a w.WM\_PAINT message.

**destroy** This method is the destructor for the `window_t` class. Because the `window_t` class is derived from `wContainer_t`, invoking this destructor will also (recursively) invoke the destructors of all the widgets held by the `window_t` object. This destructor will also free up the system brush resource (held by `_bkgBrush`) and free any storage associated with the `window_t` object. As is true for all abstract base classes, an application will not directly call this destructor method. Instead, the application will call the destructor of a derived class which, in turn, will call this destructor.

`processMessage,`

`onClose,`

`onCreate` These are private methods in the class. Applications should not call them.

`get_onPaint`

`set_onPaint` These are the accessor/mutator methods for the `onPaint` field. Applications must use these methods to access or modify the value of the `onPaint` field.

`view_t_processMessage`

Most of the time the rule is that applications do not call the `processMessage` method. In the case of the `wTabPage_t` derived class, however, there is a special case where an application needs to call the `window_t.processMessage` method. Normally, this is a somewhat difficult thing to do (especially from a `widgetProc`, from where the call is going to be made). The `view_t_processMessage` class procedure is actually an alias

for the `window_t.processMessage` method, so that a `wTabPage_t` object's `pageHandler` can call `window_t.processMessage` in that special case. In general, applications should never call `view_t.processMessage` except in this one special case. Please see the discussion of `wTabPage_t` for more details.

## 18.3.22 Timers

HOWL provides a `wTimer_t` class that lets you create "one-shot" or "periodic" timers in your HOWL applications.

### 18.3.22.1 wTimer\_t

`wTimer_t` objects are non-visual objects that can automatically call a `widgetProc` for you whenever a certain amount of time (in milliseconds) elapses. `wTimer_t` objects are useful for passing control to your code on a periodic basis even if there are no user interface interactions with your application's form.

Timers are unique amongst the widgets. You will notice that the `wTimer_t` class is derived from the `wVisual_t` class. However, `wTimer_t` objects aren't exactly visual because such objects don't appear visually on a form. `wTimer_t` objects need to be actual `window_t` objects so that they can have a Windows' `wndProc` message handler that can receive messages from a timing thread and appear in a widget list of some container object (the window itself is just a 1x1 pixel window at position (0,0) on your form). Normally, this window is hidden from the user. You should take care not to call the `show` method to make it visible.

`wTimer_t` objects operate in one of two modes: periodic and one-shot. If you initialize a `wTimer_t` object with the constant `wTimer_t.oneShot` and then start the timer, it will run for the specified amount of time (in milliseconds) and then call an `onTimeout widgetProc` exactly once. This is useful if you need exactly one notification at some future time.

`wTimer_t` objects can also operate in periodic mode. If you initialize the `wTimer_t` object with the `wTimer_t.periodic` constant, then it will automatically call an `onTimeout widgetProc` (approximately) every period milliseconds until you explicitly stop the timer.

Note that although a `wTimer_t` object spawns a thread to handling the timing chores, that thread does not call any HLA Standard Library code, and invoking the `onTimeout widgetProc` is done via a `w.PostMessage Win32` call, so you don't have to compile the program with the HLA "-thread" command-line parameter and you don't have to worry about multi-threaded synchronization. The `onTimeout` procedure always executes in the same thread as your main program.

```
wTimer_t:
    class inherits( wVisual_t );

    const
        oneShot      := 0;
        periodic     := 1;

    static
        messageCode   :dword := 0;

    var
        align( 4 );
        wTimer_private:
            record

                // Timeout value in milliseconds:

                period           :dword;

                // oneShot or periodic

                timing           :dword;

                // Widget proc to call on time out:
```

```

        onTimeOut      :widgetProc;

        // 1 = run, 0 = wait

        trigger        :dword;

        // Win32 thread handle for timer

        threadHandle    :dword;

    endrecord;

procedure create_wTimer
(
    timerName          :string;
    parentHandle        :dword;
    periodInMsec        :dword;
    timing              :dword;
    onTimeOut           :widgetProc
); external;

override method destroy;                                external;
override method processMessage;                          external;

method start;                                             external;
method stop;                                              external;

method get_onTimeOut; @returns( "eax" );                 external;
method set_onTimeOut( onTimeOut:widgetProc );            external;
method get_period; @returns( "eax" );                    external;
method set_period( period:dword );                      external;
method get_timing; @returns( "eax" );                    external;
method set_timing( timing:dword );                      external;

// Apps must never call this, it is put here for
// convenience (to be able to use "this"):

procedure _timerThread( wTimerObj:wTimer_p );           external;

endclass;

```

`wTimer_t.oneShot` This is the constant you specify as the timing argument to `create_wTimer` or `set_timing` if you want to create a one-shot timer object.

`wTimer_t.periodic`

This is the constant you specify as the timing argument to `create_wTimer` or `set_timing` if you want to create a free-running periodic timer object.

`period` This is the time, in milliseconds, that a `wTimer_t` object will delay before posting a message that will call the `onTimeOut` widgetProc. This is a private data field. Applications should not access it directly but should, instead, use the provided accessor/mutator methods.

`timing` This field determines the type of the timer. It will contain one of the following constants:

**wTimer\_t.oneShot:** the timer, when started, will call the `onTimeOut` widgetProc exactly once and then disable itself.

**wTimer\_t.periodic:** the timer, when started, will call the `onTimeOut` widgetProc about every `period` milliseconds until you explicitly stop the timer.

	This is a private data field. Applications should not access it directly but should, instead, use the provided accessor/mutator methods.
<code>onTimeOut</code>	This is the address of the widgetProc that HOWL will call when the timer times out. If this field is NULL, HOWL will not call any widgetProc and the timeout will go unnoticed by the application. This is a private data field. Applications should not access it directly but should, instead, use the provided accessor/mutator methods.
<code>_trigger</code>	This is an internal field that the <code>wTimer_t</code> object uses to communicate between the main thread and the timer thread. Applications should never access this field.
<code>_threadHandle</code>	This is an internal field that the <code>wTimer_t</code> object uses to communicate between the main thread and the timer thread. Applications should never access this field.
<code>create_wTimer</code>	<p>This is the class constructor for <code>wTimer_t</code> objects. If you call this as a class procedure (e.g., "<code>wTimer_t.create_wTimer</code>") then this procedure will allocate storage for a new <code>wTimer_t</code> object on the heap and return a pointer to that object in ESI. If you make a standard object call to this constructor, then <code>create_wTimer</code> will initialize that object in-place. Note that calling the constructor does not start the timer running. You must explicitly call the <code>wTimer_t</code> object's start method to start the timer running. This procedure has the following parameters:</p> <p><b>timerName:</b> a string specifying the timer variable's name (in HLA).</p> <p><b>parentHandle:</b> the handle of the <code>wForm</code> object on which the timer is attached.</p> <p><b>periodInMsec:</b> this is the initial timeout period for the timer. The constructor will initialize the period field with this value. You can use the accessor/mutator methods to change this value later.</p> <p><b>timing:</b> this is the initial type of the timer. This value should either be <code>wTimer_t.oneShot</code> or <code>wTimer_t.periodic</code>. You can use the accessor and mutator methods to change the timer type at a later time.</p> <p><b>onTimeOut:</b> this is either NULL or the address of a <code>widgetProc</code> that the timer will call whenever the timer times out. You can use the accessor and mutator methods to change the timeout <code>widgetProc</code> at a later time.</p>
<code>destroy</code>	This is the destructor for the <code>wTimer_t</code> object. Normally, <code>wTimer_t</code> objects are inserted onto a widget list and the container automatically calls the destructor when the container is destroyed. However, if you create a <code>wTimer_t</code> object and you don't attach it to some container's widget list, then you will need to explicitly call the destructor to free up the timer object.
<code>processMessage</code>	This is an internal procedure. Applications must not call it.
<code>get_onTimeOut,</code> <code>get_period,</code> <code>get_timing</code>	These are the accessor methods for the corresponding data fields in the class. Applications should call these methods to access the data fields rather than reading their values directly.
<code>set_onTimeOut,</code> <code>set_period,</code> <code>set_timing</code>	These are the mutator methods for the corresponding data fields in the class. Applications should call these methods to access the data fields rather than writing their values directly. Note that you should take care when setting these values while a timer is actually running. Although access to these objects is synchronized (you don't have to worry about thread problems), changing these values while a timer is operating can make your programs difficult to read and modify and it can introduce some obscure bugs.
<code>start</code>	This starts the timer. If the timer was already running, this will kill the current timer thread and start a new one. When the timer expires (after no sooner than <code>period</code> milliseconds), the timer thread will post a message to the <code>wTimer_t</code> object to tell it to call the <code>onTimeOut</code> <code>widgetProc</code> . Note that the <code>onTimeOut</code> procedure could actually be called much later, based on the number of messages in the Windows' message queue and the

time it takes to process all those messages. Do not assume that exactly `period` milliseconds have passed since the call to `start` when your `onTimeout` procedure begins execution. It could actually be much later than this.

`stop`

This method immediately stops the timer. This generally means that (if the timer is running) there will be no call to the `onTimeout` procedure. However, the timer could have already posted an `onTimeout` call in the message queue, so you should not assume that there will be no call to `onTimeout` after you call `stop`.



## 19 The Linux Module (linux.hhf)

The hll.hhf library module adds a switch/case/default/endswitch statement that is similar to the Pascal case statement and the C/C++ switch statement.

### 19.1 The Linux Module

To use the Linux functions in your application, you will need to include one of the following statements at the beginning of your HLA application:

```
#include( "linux.hhf" )
```

Note that including stdlib.hhf does not automatically include the linux.hhf header file. You must explicitly include the linux.hhf header file to make use of its functionality.

### 19.2 The Linux Header File

The Linux module contains constants, data types, procedure prototypes, and other declarations needed to make Linux system calls. Obviously, only HLA/Linux users should be making calls to this module. **Warning:** it is perfectly possible to compile the Linux module under Windows and attempt to run the resulting code. However, this will surely crash the system.

Linux systems calls are made via the INT(\$80) instruction. The HLA Linux module provides wrappers for all these calls so you can invoke them using a high level syntax. Calling the HLA Linux wrappers is a much better idea than embedding INT(\$80) invocations directly in your code. Sometimes the Linux system calls change (hopefully for the better). Although the Linux developers have done a good job of maintaining older calls in the kernel, your programs that make these older calls may not benefit from additional functionality added to the kernel. On the other hand, if all your programs call the HLA wrapper routines for Linux, then you've only got to change the call in one location (in the wrapper), rather than throughout all your projects, whenever a system call changes.

This section will not attempt to document each of the Linux calls that HLA's Linux module provides. You can read all about these functions in Linux use the "man -S 2 *function\_name*" command. The calling sequence is (usually) identical to the "C" interface, so there is no need to regurgitate that information here. Because C and HLA have different sets of reserved words, there were a few conflicts between the standard (C-based) function names and the names that HLA uses, this section will elaborate on those. Also, C's structs and functions use a different namespace and the Linux (UNIX) kernel programmers have employed the dubious style of using the same name for functions and structures. Since HLA doesn't allow this, some type names have been changed, as well. Finally, to prevent namespace pollution in HLA programs, HLA actually uses a NAMESPACE to hold the Linux identifiers (much like other library modules in HLA), so common Linux function names and datatypes will require a "linux." prefix.

It is not good programming style to use all uppercase within identifiers (despite the long-standing C/Unix tradition). Therefore, most all-uppercase constant identifiers you'd normally find in a Linux header file use the HLA convention of lowercase (which is easier to read). Generally, when there is a conflict between a C identifier and an HLA reserved word, the conflict is resolved by prepending an underscore. For example, the Linux system call "exit" becomes "\_exit" in HLA (since exit is an HLA reserved word). The two common exceptions to this rule are for the identifiers "name" and "value" (both HLA reserved words) which are usually converted to "theName" or "theValue" in the linux.hhf header file.

Whenever a type name conflicts with a procedure name, the linux.hhf header file appends "\_t" to the type name. This is a common Unix practice and one wonders why these structure names didn't have the "\_t" suffix to begin with.

Certain Linux functions are overloaded allowing one, two, or possibly three parameters. The linux.hhf header file contains two or three prototypes for each function, each with a fixed number of parameters. However, you can still call the functions using the standard C syntax because HLA provides macros to simulate function overloading (i.e., a variable number of parameters) for these particular functions. Generally, the macro uses the standard C/Linux name (e.g. linux.sysfs) while the actual HLA procedures use a numeric suffix to denote the number of parameters (e.g., linux.sysfs1, linux.sysfs2, and linux.sysfs3).

Please see the "linux.hhf" header file for more details on the spelling of Linux constants, types, and function names.

You should also note that there is a list of constants that begin with "sys\_" at the end of the header file. These are the function opcodes that one passes in EAX to the INT(\$80) system call. If you're going to make the INT(\$80) calls directly yourself, you should, at least, use these symbol names (e.g., "sys\_exit").

## 20 Lists Module (lists.hhf)

The list.bodyhhf library module provides a class data type and a set of functions to manipulate linked lists within a program.

### 20.1 The Lists Module

To use the list functions in your application, you will need to include one of the following statements at the beginning of your HLA application:

```
#include( "lists.hhf" )
or
#include( "stdlib.hhf" )
```

#### 20.1.0.1 List Data Types

The HLA Standard Library provides a generic list abstract data type via the lists module. The lists module provides two classes: a generic list class and a generic, abstract, *node\_t* class. These classes have (approximately) the following definitions:

```
nodePtr_t :pointer to node_t;
node_t:
class

    var
        Prev: pointer to node_t;
        Next: pointer to node_t;

    procedure create; @returns( "esi" ); @external;
    method destroy; @abstract;
    method cmpNodes( n:nodePtr_t ); @abstract;

endclass;

list_t:
class

    var
        Head: pointer to node_t;
        Tail: pointer to node_t;
        Cnt: uns32;
        align(4);

    procedure create; @returns( "esi" );
    method destroy;
    method numNodes; @returns( "eax" );
    method append_index( var n:node_t; posn: dword );
        @returns( "esi" );

    method append_node( var n:node_t; var after: node_t );
        @returns( "esi" );

    method append_last( var n:node_t ); @returns( "esi" );
    method insert_index( var n:node_t; posn:dword ); @returns( "esi" );
    method insert_node( var n:node_t; var before:node_t );
        @returns( "esi" );

    method insert_first( var n:node_t ); @returns( "esi" );
```

```

method delete_index( posn:dword );
method delete_node( var n:node_t );
method delete_first;
method delete_last;
method index( posn:dword );
method xchgNodes( n1:nodePtr_t; n2:nodePtr_t );
method sort;
method reverse;
method search( cmpThunk:thunk );
iterator nodeInList;
iterator nodeInListReversed;
iterator filteredNodeInList( t:thunk );
iterator filteredNodeInListReversed( t:thunk );

endclass;

```

The *node\_t* class is an abstract base class from which you must derive a node type for the nodes in your list. You would normally override the *node\_t.create* procedure and write a procedure that specifically allocates storage for an object of type *node\_t* and initializes any important data fields. If you like, your overloaded *create* procedure can call *node\_t.create* to initialize the link fields of the node you create, although this is not strictly necessary.

The *node\_t.destroy* method is an abstract method that you must override. The *list\_t.destroy* method calls *node\_t.destroy* (or, at least, your overloaded version of it) in order to free the storage associated with a given node. A typical concrete implementation of this function looks like the following:

```

method MyNode.destroy; @nodisplay; @noframe;
begin destroy;

    // On entry, ESI points at the current node object.
    // Free the storage associated with this node.

    if( isInHeap( esi )) then

        free( esi );

    endif;

end destroy;

```

The *node\_t.cmpNodes* method is another abstract method you may need to override. This method compares the current node (referenced by *this*) against the node whose address the caller passes as the single argument. This method compares the two nodes and sets the carry and zero flags in a manner consistent with an unsigned integer comparison (that is, it sets the carry flag if the *this* node is less than the parameter node; it sets the zero flag if the two nodes are equal; it clears these two flags if the opposite conditions hold). The *list\_t.sort* and *list\_t.search* functions use *node\_t.cmpNodes*; if you use either of these functions in the *list\_t* objects you create, you will need to provide a concrete implementation of the *node\_t.cmpNodes* method. Note that because *node\_t.cmpNodes* is an abstract method, there is no default implementation for this function – you must provide a concrete implementation if you call it or you call some other function that calls it. Here is a sample implementation that demonstrates this:

```

method MyNode.cmpNodes;
var
    thisSave
begin cmpNodes;

    // Assume there is a "keyID" signed integer field in MyNode and
    // when we compare the two nodes we simply compare the int32 values
    // and set the flags for an unsigned comparison.

    push( eax );

```

```

mov( n, eax );
mov( (type MyNode [eax]).keyID, eax );
cmp( eax, this.keyID );
if( @1 ) then

    stc();// Make @b. Note that Z is clear

else

    clc();// Make @nb.

    // Note that Z is set appropriately at this point.

endif;
pop( eax );

end cmpNodes;

```

For a typical example of an overloaded `node_t` class, see the `listDemo.hla` example in the HLA examples subdirectory.

The `list_t` class is an abstract data type used to maintain lists of nodes. Internally, the `list_t` class represents lists of nodes using a doubly-linked list, although your applications should not be aware of the internal implementation. Likewise, for efficiency reasons the `list_t` class maintains a pointer to the head of the list, a pointer to the tail of the list, and a count of the number of nodes currently in the list. Your applications should ignore these fields (note that you can obtain the number of nodes in the list by calling the `numNodes` method) and treat the fields as private to the class.

## 20.2 List\_t Class Function Types

In most HLA classes, there are three types of functions: (static) procedures, (dynamic), and (dynamic) iterators. The only difference between a method and a procedure is how the program actually calls the function: the program calls procedures directly, it calls methods indirectly through an entry in the virtual method table (VMT). Static procedure calls are very efficient, but you lose the benefits of inheritance and functional polymorphism when you define a function as a static procedure in a class. Methods, on the other hand, fully support polymorphic calls, but introduce some efficiency issues.

First of all, unlike static procedures, your program will link in all methods defined in your program *even if you don't explicitly call those methods in your program*. Because the call is indirect, there really is no way for the assembler and linker to determine whether you've actually called the function, so it must assume that you do call it and links in the code for each method in the class. This can make your program a little larger because it may be including several date class functions that you don't actually call.

The second efficiency issue concerning method calls is that they use the EDI register to make the indirect call (static procedure calls do not disturb the value in EDI). Therefore, you must ensure that EDI is free and available before making a virtual method call, or take the effort to preserve EDI's value across such a call.

A third, though extremely minor, efficiency issue concerning methods is that the class' VMT will need an extra entry in the virtual method table. As this is only four bytes per class (not per object), this probably isn't much of a concern.

The HLA Standard Library predefines two classes: `list_t` and `virtualList_t`. They differ in how they define the functions appearing in the class types. The `list_t` type uses static procedures for all functions, the `virtualList_t` type uses methods for all class functions. Therefore, `list_t` objects will make direct calls to all the functions (and only link in the procedures you actually call); however, `list_t` objects do not support function polymorphism in derived classes. The `virtualList_t` type does support polymorphism for all the class methods, but whenever you use this data type you will link in all the methods (even if you don't call them all) and calls to these methods will require the use of the EDI register.

It is important to understand that `list_t` and `virtualList_t` are two separate types. Neither is derived from the other. Nor are the two types compatible with one another. You should take care not to confuse objects of these two types if you're using both types in the same program (better yet, don't use both types in the same program – use `virtualList_t` if you need polymorphism).

## 20.3 Creating New List Class Types

As it turns out, the only difference between a method and a procedure (in HLA) is how that method/procedure is called. The actual function code is identical regardless of the declaration (the reason HLA supports method and procedure declarations is so that it can determine how to populate the VMT and to determine how to call the function). By pulling some tricks, it's quite possible to call a procedure using the method invocation scheme or call a method using a direct call (like a static procedure). The Standard Library list class module takes advantage of this trick to make it possible to create new list classes with a user-selectable set of procedures and methods. This allows you to create a custom list type that uses methods for those functions you want to override (as methods) and use procedures for those functions you don't call or will never override (as virtual methods). Indeed, the *list\_t* and *virtualList\_t* data types were created using this technique. The *list\_t* data type was created specifying all functions as procedures, the *virtualList\_t* data type was created specifying all functions as methods. By using the *\_hla.make\_listClass* macro, you can create new data types that have any combination of procedures and methods.

```
_hla.make_listClass( className, "<list of methods>" )
```

*\_hla.make\_listClass* is a macro that generates a new data type. As such, you should only invoke this macro in an HLA type declaration section. This macro requires two arguments: a class name and a string containing the list of methods to use in the new data type. The method list string must contain a sequence of method names (typically separated by spaces, though this isn't strictly necessary) from the following list:

```
destroy
numNodes
appendIndex
appendNode
appendLast
insertIndex
insertNode
insertFirst
deleteIndex
deleteNode
deleteFirst
deleteFast
index
xchgNodes
sort
reverse
search
```

Here is *\_hla.make\_listClass* macro invocation that creates the *virtualList* type:

```
type
_hla.make_listClass
(
    virtualList_t,
    "destroy"
    "numNodes"
    "appendIndex"
    "appendNode"
    "appendLast"
    "insertIndex"
    "insertNode"
    "insertFirst"
    "deleteIndex"
    "deleteNode"
    "deleteFirst"
    "deleteFast"
    "index"
    "xchgNodes"
    "sort"
    "reverse"
    "search"
);
```

(For those unfamiliar with the syntax, HLA automatically concatenates string literals that are separated by nothing but whitespace; therefore, this macro contains exactly two arguments, the *virtualList\_t* name and a single string containing the concatenation of all the strings above.)

From this macro invocation, HLA creates a new data type using methods for each of the names appearing in the string argument. If a particular date function's name is not present in the *\_hla.make\_listClass* macro invocation, then HLA creates a static procedure for that function. As a second example, consider the declaration of the *list\_t* data type (which uses static procedures for all the list functions):

```
type
    _hla.make_listClass( list_t, " " );
```

Because the function string does not contain any of the list function names, the *\_hla.make\_listClass* macro generates static procedures for all the list functions.

The *list\_t* type is great if you don't need to create a derived list class that allows you to polymorphically override any of the list functions. If you do need to create methods for certain functions and you don't mind linking in all the list class functions (and you don't mind the extra overhead of a method call, even for those functions you're not overloading), the *virtualList\_t* data type is convenient to use because it makes all the functions virtual (that is, methods). Probably 99% of the time you won't be calling the list functions very often, so the overhead of using method invocations for all list functions is irrelevant. In those rare cases where you do need to support polymorphism for a few list functions but don't want to link in the entire set of list functions, or you don't want to pay the overhead for indirect calls to functions that are never polymorphic, you can create a new list class type that specifies exactly which functions require polymorphism.

For example, if you want to create a date class that overrides the definition of the sort and search functions, you could declare that new type thusly:

```
type
    _hla.make_listClass
    (
        MyListClass,
        "sort"
        "search"
    );
```

This new class type (*MyListClass*) has two methods, *sort* and *search*, and all the other list functions are static procedures. This allows you to create a derived class that overloads the *sort* and *search* methods and access those methods when using a generic *MyListClass* pointer, e.g.,

```
type
    derivedMyListClass :
        class inherits( MyListClass );

        override method sort;
        override method search;

    endclass;
```

Again, it is important for you to understand that types created by *\_hla.make\_listClass* are base types. They are not derived from any other class (e.g., *virtualList* is not derived from *list* or vice-versa). The types created by the *\_hla.make\_listClass* macro are independent and incompatible types. For this reason, you should avoid using different base list class types in your program. Pick (or create) a base list class and use that one exclusively in an application. You'll avoid confusion by following this rule.

## 20.4 List Procedures, Methods, and Iterators

Because you can create your own list data types, describing list functions as procedures or methods is somewhat inaccurate. In the sections that follow, a function is described as a "procedure" if it is always a static procedure and you cannot override that (this only applies to the constructor); a function is described as a "method" if you can create a new data type and define that function to be a static procedure or a dynamic method via the *\_hla.make\_listClass* macro. Note that the four iterators defined in the list class (*list\_t.nodeInList*,

*list\_t.nodeInListReversed*, *list\_t.filteredNodeInList*, and *list\_t.filteredNodeInListReversed*) are always dynamic iterators, you cannot change their definition.

As is typical for the Standard Library documentation when describing classes and objects, this chapter does not provide any examples of low-level assembly language calls to the various methods in the *list\_t* class. The assumption here is that someone who is doing object-oriented programming in assembly language is perfectly happy using the high-level method calls (particularly as the low-level method invocations are rather messy). If you're an exception to this rule, please consult the HLA documentation for details on making direct (low-level) calls to class methods and iterators.

The calling sequence examples appearing throughout this chapter use the following object declarations:

```
static
  sList:virtualList_t;
  pList:pointer to virtualList_t;
```

Note that the calling sequences are exactly the same for static and virtual objects. That is, you could replace the two *virtualList\_t* data types above with *list\_t* and the examples would all still be syntactically correct.

When discussing methods, the following sections claim that any call to a method will wipe out the value in the EDI register. This is true if the class data type actually uses methods. If you've created a new list data type using *\_hla.make\_listClass* and you've defined a function to be a procedure rather than a method, then the call is direct and it does not necessarily disturb the value of the EDI register. However, you should not make this assumption. Some methods might actually assume that it's okay to disturb the value in EDI as it was used to hold the VMT address for the call. Better safe than sorry – assume that if it's a method, EDI's value gets modified.

## 20.5 List Constructor and Destructor

```
procedure list_t.create; @returns( "esi" );
```

This is the standard constructor for the list class. If you call this class procedure via *list\_t.create()* it will allocate storage for a new *list\_t* object, initialize the fields of that object (to the empty list), and return a pointer to that *list\_t* object in ESI. If you call this class procedure via *someListVarName.create()* then this procedure will initialize the (presumably) allocated *list\_t* object (again, to the empty list).

HLA high-level calling sequence examples:

```
// Constructor call that allocates storage for a list object:
```

```
virtualList_t.create();
mov( esi, pList );
```

```
// Constructor call that initializes an already-allocated object:
```

```
sList.create();
```

```
method list_t.destroy;
```

This method frees the storage associated with each node in the list (if the individual nodes were allocated on the heap), it then frees the storage associated with the *list\_t* object itself, assuming the list was allocated on the heap. Note that successful execution of this method requires that you create a derived class from the abstract base class *node\_t* and that you've overridden the *node\_t.destroy* method. The *list\_t.destroy* method deallocates the nodes in the list by calling the *node\_t.destroy* method for each node in the list.

HLA high-level calling sequence examples:

```
sList.destroy();
pList.destroy();
```



## 20.6 Accessor Functions

```
method list_t.numNodes; @returns( "eax" );
```

This function returns the number of nodes currently in the list in the EAX register. You should always call this routine rather than access the *list\_t.Cnt* field directly.

HLA high-level calling sequence examples:

```
sList.numNodes();
mov( eax, sNumNodes );

pList.numNodes();
mov( eax, pNumNodes );
```

### 20.6.0.1 Adding Nodes to a List

```
#macro list_t.append( node, posn );
#macro list_t.append( node, node );
#macro list_t.append( node );
```

The *list\_t.append* macro provides function overloading on the *list\_t.append\_index*, *list\_t.append\_node*, and *list\_t.appendLast* functions. The *list\_t.append* macro checks the number and type of the parameters and calls the appropriate *list\_t.append\_\** function whose signature matches the argument list. See the discussion of the following three methods for details on the specific calls.

```
method list_t.append_index( var n:node_t; posn: dword ); @returns( "esi" );
```

This method appends node *n* to the list after node *posn* in the list. If *posn* is greater than or equal to the number of nodes in the list, then this method appends node *n* to the end of the list. Normally, you would not call this method directly. Instead, you would use the

```
sList.append(n, posn);
```

macro to call this method. This function returns a pointer to node *n* in ESI. As with most method invocations, this call wipes out the value in EDI.

HLA high-level calling sequence examples:

```
sList.append_index( MyNodePtr, 4 );// Append after fifth node
pList.append( MyNodePtr, 5 );// Append after sixth node
```

```
method list_t.append_node( var n:node_t; var after: node_t );
@returns( "esi" );
```

This method inserts node *n* in the object list immediately after node *after* in that list. This method assumes that *after* is a node in the object's list; it does not validate this fact. Therefore, you must ensure that *after* is a member of the object's list. Normally, you would not call this function directly; instead, you would invoke the

```
listVar.append( n, after );
```

macro to do the work. This function returns a pointer to node *n* in ESI. As with most method invocations, this call wipes out the value in EDI.

HLA high-level calling sequence examples:

```
// Append NewNode after the NodeInList node:

sList.append_node( NewNode, NodeInList );

// Append anotherNewNode after someNodeInpList:

pList.append( AnotherNewNode, someNodeInpList );
```

**method list\_t.append\_last( var n:node\_t ); @returns( "esi" );**

This method appends node *n* to the end of the object list. Normally you would not call this method directly, instead you would just invoke the macro:

```
listVar.append(n);
```

This function returns a pointer to node *n* in ESI. As with most method invocations, this call wipes out the value in EDI.

HLA high-level calling sequence examples:

```
// Append NewNode at the end of the list:

sList.append_last( NewNode );

// Append anotherNewNode at the end of the pList:

pList.append( AnotherNewNode );
```

```
#macro list_t.insert( node, posn );
#macro list_t.insert( node, node );
#macro list_t.insert( node );
```

The *list\_t.insert* macro provides function overloading on the *list\_t.insert\_index*, *list\_t.insert\_node*, and *list\_t.insertFirst* functions. The *list\_t.insert* macro checks the number and type of the parameters and calls the appropriate *list\_t.insert\_\** function whose signature matches the argument list. See the discussion of the following three methods for details on the specific calls.

**method list\_t.insert\_index( var n:node\_t; posn:dword ); @returns( "esi" );**

This method inserts node *n* before the *posn*<sup>th</sup> node in the list. If *posn* is greater than or equal to the number of nodes in the list, this method simply appends the node to the end of the list (remember, nodes are numbered from 0..Cnt-1; so if *posn*=*Cnt* then that would imply inserting the node at the end of the list). Normally you would not call this method directly; instead, you'll invoke the

```
listVar.insert( n, posn);
```

macro to do the job. This function returns a pointer to node *n* in ESI. As with most method invocations, this call wipes out the value in EDI.

HLA high-level calling sequence examples:

```
sList.insert_index( MyNodePtr, 4 );// Insert before fifth node
pList.insert( MyNodePtr, 5 );// Insert before sixth node
```

```
method list_t.insert_node( var n:node_t; var before:node_t );
    @returns( "esi" );
```

This method inserts node *n* before node *before* in the object's list. This method assumes that *before* is an actual member of the list, it does not verify this prior to insertion. You would not normally call this routine directly. Instead, invoke the *listVar.insert( n, before )* macro to do the actual work. This function returns a pointer to node *n* in ESI. As with most method invocations, this call wipes out the value in EDI.

HLA high-level calling sequence examples:

```
// Insert NewNode before the NodeInList node:

sList.insert_node( NewNode, NodeInList );

// Insert anotherNewNode before someNodeInpList:

pList.insert( AnotherNewNode, someNodeInpList );
```

```
method list_t.insert_first( var n:node_t ); @returns( "esi" );
```

This function inserts node *n* at the beginning of the object's list. You would not normally call this method directly; you should normally invoke the *listVar.insert( n )* macro and let it do all the work. This function returns a pointer to node *n* in ESI. As with most method invocations, this call wipes out the value in EDI.

HLA high-level calling sequence examples:

```
// Insert NewNode at the beginning of the list:

sList.insert_last( NewNode );

// Insert anotherNewNode at the start of the pList:

pList.insert( AnotherNewNode );
```

## 20.7 Removing Nodes From a List

```
#macro list_t.delete( posn );
#macro list_t.delete( node );
#macro list_t.delete();
```

These macros overload the *list\_t.delete\_index*, *list\_t.delete\_node*, and *list\_t.delete\_first* methods in the list class. The macro determines which of these methods to call by testing the number and types of the macro's arguments. Note that this macro does not overload the *list\_t.delete\_last* method as it does not have a unique signature (i.e., *list\_t.delete\_last*'s signature would be identical to *list\_t.delete\_first*'s).

```
method list_t.delete_index( posn:dword ); @returns( "esi" );
```

This method removes the *posn<sup>th</sup>* node from the list and returns a pointer to this node in ESI. Normally you would invoke the

```
list_t.delete( posn );
```

macro rather than calling this method directly.

HLA high-level calling sequence examples:

```
sList.delete_index( 4 );// Delete the fifth node
pList.insert( ecx );// Delete the node whose index is in ECX
```

**method list\_t.delete\_node( var n:node\_t ); @returns( "esi" );**

This method removes node *n* from the list and returns a pointer to this node in ESI. This method assumes that node *n* actually is in the list; it does not verify this. Normally, you would invoke the

```
list_t.delete(n);
```

macro rather than call this method directly.

HLA high-level calling sequence examples:

```
// Delete the deleteMe node:

sList.delete_node( deleteMe );
mov( esi, deleted_node );

// Delete anotherUselessNode:

pList.delete( anotherUselessNode );
mov( esi, deleted_node_too );
```

**method list\_t.delete\_first; @returns( "esi" );**

This method removes the first node from the list and returns a pointer to this node in ESI. Normally you would not call this method directly but you would invoke the

```
list_t.delete();
```

macro instead.

HLA high-level calling sequence examples:

```
// Delete the node at the beginning of the list:

sList.delete_first();
mov( esi, deleted_node );

pList.delete();
mov( esi, deleted_node_too );
```

**method list\_t.delete\_last; @returns( "esi" );**

This method removes the last node from the list and returns a pointer to this node in ESI.

HLA high-level calling sequence examples:

```
// Delete the node at the end of the list:

sList.delete_last();
mov( esi, deleted_node );
```

## 20.8 Accessing Nodes in a List

```
method list_t.index( posn:dword ); @returns( "esi" );
```

This method returns a pointer to the *posn*<sup>th</sup> node in the list in the ESI register. It returns NULL if the list is empty. It returns the address of the last node in the list if *posn* >= *Cnt*.

HLA high-level calling sequence examples:

```
// Access the 5th node in the list:

sList.index( 4 );
mov( esi, fifth_node );
```

```
iterator list_t.nodeInList;
```

This iterator returns a pointer to each node in a list in the ESI register. This iterator traverses the list forward – from the beginning of the list to the end of the list. Like most iterators, you normally use this iterator within a FOREACH loop.

HLA high-level calling sequence examples:

```
// Traverse the entire list:

foreach sList.nodeInList() do

    << Do something with the node pointer in ESI... >>

endfor;

foreach pList.nodeInList() do

    << Do something with the node pointer in ESI... >>

endfor;
```

```
iterator list_t.nodeInListReversed;
```

This iterator returns a pointer to each node in a list in the ESI register. This iterator traverses the list backward, from the end of the list to the beginning of the list. Like most iterators, you normally use this iterator within a FOREACH loop.

HLA high-level calling sequence examples:

```
// Traverse the entire list backwards:

foreach sList.nodeInListReversed() do

    << Do something with the node pointer in ESI... >>

endfor;

foreach pList.nodeInListReversed() do

    << Do something with the node pointer in ESI... >>
```

```
endfor;
```

```
iterator list_t.filteredNodeInList( t:thunk );
```

This iterator traverses the list and returns a pointer to each node that is "approved" by the thunk *t*. This iterator traverses the list forward – from the beginning of the list to the end of the list. Like most iterators, you normally use this iterator within a FOREACH loop.

On each FOREACH loop iteration, the *list\_t.filteredNodeInList* iterator will call the *t* thunk and pass it a pointer to the current node in ESI. The (caller-defined) thunk will test that node (application-specific) and return true in AL if the FOREACH loop should iterate on that particular node; the thunk should return false in AL if the FOREACH loop should skip that particular node in the iteration sequence.

HLA high-level calling sequence examples:

```
// Traverse the list and operate on all nodes whose
// "j" field is greater than or equal to 10:
```

```
foreach
  sList.filteredNodeInList
  (
    thunk
    #{
      xor( eax, eax );
      cmp( (type MyNode [esi]).j, 10 );
      setae( al );
    }#
  )
do

  << Do something with the node pointer in ESI... >>

endfor;
```

```
// Traverse the list and operate on all nodes whose
// "j" field is equal to the "k" field:
```

```
foreach
  sList.filteredNodeInList
  (
    thunk
    #{
      mov( (type MyNode [esi]).j, eax );
      cmp( eax, (type MyNode [esi]).k );
      mov( 0, eax );
      sete( al );
    }#
  )
do

  << Do something with the node pointer in ESI... >>

endfor;
```

```
iterator list_t.filteredNodeInListReversed( t:thunk );
```

This iterator behaves just like *list\_t.filteredNodeInList* except that it traverses the list backwards. As for *list\_t.filteredNodeInList*, you must provide a thunk that approves or rejects each node in the list. Only approved

nodes are passed along to the body of the FOREACH loop. Like most iterators, you normally use this iterator within a FOREACH loop.

HLA high-level calling sequence examples:

```
// Traverse the list and operate on all nodes whose
// "j" field is greater than or equal to 10:
```

```
foreach
  sList.filteredNodeInListReversed
  (
    thunk
    #{
      xor( eax, eax );
      cmp( (type MyNode [esi]).j, 10 );
      setae( al );
    }#
  )
do

  << Do something with the node pointer in ESI... >>

endfor;
```

```
// Traverse the list and operate on all nodes whose
// "j" field is equal to the "k" field:
```

```
foreach
  sList.filteredNodeInListReversed
  (
    thunk
    #{
      mov( (type MyNode [esi]).j, eax );
      cmp( eax, (type MyNode [esi]).k );
      mov( 0, eax );
      sete( al );
    }#
  )
do

  << Do something with the node pointer in ESI... >>

endfor;
```

## 20.9 Miscellaneous List Functions

**method list\_t.reverse;**

This method reverses the nodes in the list. That is, the first node becomes the last node, the second node becomes the second to the last node, ..., and the last node becomes the first node. This function only changes the (private) Next and Prev pointers in each node, it does not physically move the nodes around in memory (so pointers to the nodes remain valid) nor does it change any other data in the nodes in the list.

HLA high-level calling sequence examples:

```
// Reverse the lists:
```

```
sList.reverse();
pList.reverse();
```

```
method list_t.xchgNodes( n1:nodePtr_t; n2:nodePtr_t );
```

This method exchanges two nodes in the list. Specifically, this function swaps the nodes' (private) *Next* and *Prev* fields and, if necessary, updates the beginning and ending node pointers in the list object. The *n1* and *n2* parameters must contain pointers to nodes within the list or this function will produce undefined results. This function does not check *n1* or *n2* to verify that they are within the list, it is the caller's responsibility to ensure this.

HLA high-level calling sequence examples:

```
// Exchange the 4th and 7th nodes in the list:

sList.index( 3 );
mov( eax, ebx ); // Get address of the 4th node.
sList.index( 6 ); // Get the address of the 7th node.
sList.xchgNodes( eax, ebx ); // Exchange the two nodes.
```

```
method list_t.sort;
```

This method sorts the nodes in the list in ascending order. This function invokes the *node\_t* class *cmpNodes* method in order to sort the list, so if you use this function you must provide a concrete implementation of the *cmpNodes* method or the HLA run-time system will raise an "Abstract Method Executed" exception.

Note that if you want to sort the list in descending order, and you don't otherwise need to sort it in ascending order, you can define the *node\_t.cmpNodes* method to reverse the state of the carry flag (that is, return carry set on greater than and carry clear on less than or equal). However, be careful if you do this as those semantics will exist for all lists that try to sort the particular *node\_t* class you've defined this way. Perhaps a better solution would be to overload the list class you've defined and create a new *sort* procedure that sorts the data in descending order. Or simply modify the list class source code and add a *list\_t.sortReverse* function.

HLA high-level calling sequence examples:

```
// Sort the lists:

sList.sort();
pList.sort();
```

```
method list_t.search( cmpThunk:thunk ); @returns( "eax" );
```

This method searches for a specific node in the list (starting at the front of the list and sequentially searching toward the end of the list). It executes the *cmpThunk* thunk on each node until either the thunk returns true in AL (that is, it does not return false in AL) or it reaches the end of the list. If the thunk ever returns true, then this function returns a pointer to the associated node in the EAX register. If the search function scans the entire list and *cmpThunk* always returns false, then this function will return NULL (zero) in EAX upon reaching the end of the list.

This function is very similar to the *list\_t.filteredNodeInList* function except that it does not iterate on every matching node in the list. Instead, this function returns only the first matching occurrence found in the list.

HLA high-level calling sequence examples:

```
// Search for the first node whose
// "j" field is equal to 10:
```



```
sList.search
(
    thunk
    #{
        xor( eax, eax );
        cmp( (type MyNode [esi]).j, 10 );
        sete( al );
    }#
);
if( eax <> NULL ) then

    << do something with the node pointed at by EAX... >

endif;
```



## 21 Math Module (math.hhf)

The HLA math library module provides several large integer arithmetic/logical, trigonometric, and logarithmic routines that extend those provided directly in the CPU and FPU. Note that many of the transcendental functions place strict limits on the values of their parameters. See a reasonable math text or the Intel documentation for details.

**A Note About the FPU:** The Standard Library code (and the math module in particular) makes use of the FPU when computing certain mathematical functions. You should exercise caution when using MMX instructions in a program that makes use of the Standard Library. In particular, you should ensure that you are always in FPU mode (by executing an EMMS instruction) after you are finished using MMX instructions. Better yet, you should avoid the MMX instruction set altogether and use the improved SSE instruction set that accomplishes the same tasks (and doesn't disturb the FPU).

### 21.1 The Math Module

To use the math functions in your application, you will need to include one of the following statements at the beginning of your HLA application:

```
#include( "math.hhf" )
or
#include( "stdlib.hhf" )
```

### 21.2 Math Data Types

The math module of the HLA standard library works with seven different data types: 64-bit integers (signed or unsigned), 128-bit integers (signed or unsigned), and real values (32-bit, 64-bit, and 80 bits). Each function specifies the operand data types it expects to use.

### 21.3 64-Bit Arithmetic and Logical Operations

The HLA Standard Library provides a complete set of arithmetic and logical operations for 64-bit integers. Extended precision arithmetic (especially 64-bit) is fairly straight forward and an in-line coding of most of these functions will generally be faster than calling these functions. For example, a full 64-bit extended precision addition requires about the same number of instructions as it takes to simply pass the parameters to the *math.addq* routine. Therefore, do not call these routines if performance is important, use in-line code instead.

Another reason (beyond the procedure call overhead) that these procedures are slower than the in-line code is that the standard extended precision add sequence does not set the zero flag properly; these procedures have to execute several additional instructions to preserve the carry, sign, and overflow flags as well as properly set the zero flag. So, for example, if you don't use the value of the zero flag upon return, all this extra work goes to waste.

These procedures are convenient to use and are perfectly acceptable when performance is not an issue. Another advantage is that these routines work memory to memory and don't disturb the values in any registers; and also, these routines use a "three-address" form that allows a different destination address (i.e., the destination does not have to be the same as one of the source operands. Of course, as already mentioned, these functions tend to set the flags the same way the basic machine instructions would set them, a big advantage if you are testing the flags after extended-precision arithmetic operations.

```
math.addq( left:qword; right:qword; var dest:qword );
```

This routine adds two quad-word 64-bit integer values and stores the result in a 64-bit destination memory location. The values may be signed or unsigned. This routine computes the following:

```
dest := left + right;
```

This function sets the 80x86 flags exactly the same way that the standard ADD instruction does. In particular, you may test the zero flag afterwards for a zero result, you can test the sign flag to determine if there was a negative result, the carry and overflow flags denote unsigned or signed overflow (respectively).

HLA high-level calling sequence example:

```
// Compute Sum64 := i64 + j64:

math.addq( i64, j64, Sum64 );
```

HLA low-level calling sequence example:

```
// Compute Sum64 := i64 + j64:

push((type dword i64[4]));
push((type dword i64));
push((type dword j64[4]));
push((type dword j64));
lea( eax, Sum64 );
push( eax );
call math.addq;
```

**math.subq( left:qword; right:qword; var dest:qword );**

This function subtracts two 64-bit integer values and stores their difference in *dest*. The values may be signed or unsigned. This function computes the following:

```
dest := left - right;
```

This function sets the 80x86 flags exactly the same way that the standard SUB instruction does. In particular, you may test the zero flag afterwards for a zero result, you can test the sign flag to determine if there was a negative result, the carry and overflow flags denote unsigned or signed overflow (respectively).

HLA high-level calling sequence example:

```
// Compute Dif64 := i64 - j64:

math.subq( i64, j64, Dif64 );
```

HLA low-level calling sequence example:

```
// Compute Dif64 := i64 - j64:

push((type dword i64[4]));
push((type dword i64));
push((type dword j64[4]));
push((type dword j64));
lea( eax, Dif64 );
push( eax );
call math.subq;
```

**math.divq( left:qword; right:qword; var dest:qword );**

This routine divides one unsigned 64-bit value by another. It computes the following:

```
dest := left div right;
```

Since the 80x86 flags don't contain useful values after the execution of the DIV instruction, these routines also leave the flags scrambled and you can't count on flag values upon return. This routines will raise an *ex.DivideError* exception if you attempt a division by zero.

Note that the discussion of the triviality of 64-bit arithmetic does not apply to division. Calling this function (versus attempting an in-line implementation) is reasonably efficient and you needn't heed the warning given earlier.

Also note that this function does not automatically compute the remainder (as the x86 DIV instruction does). The math module provides the *math.modq* function if you need to compute the remainder of the division of two 64-bit values.

HLA high-level calling sequence example:

```
// Compute Quo64 := i64 div j64:

math.divq( i64, j64, Quo64 );
```

HLA low-level calling sequence example:

```
// Compute Quo64 := i64 div j64:

push((type dword i64[4]));
push((type dword i64));
push((type dword j64[4]));
push((type dword j64));
lea( eax, Quo64 );
push( eax );
call math.divq;
```

**math.idivq( left:qword; right:qword; var dest:qword );**

This routine divides one signed 64-bit value by another. It computes the following:

```
dest := left idiv right;
```

Since the 80x86 flags don't contain useful values after the execution of the IDIV instruction, these routines also leave the flags scrambled and you can't count on flag values upon return. This routines will raise an *ex.DivideError* exception if you attempt a division by zero.

Note that the discussion of the triviality of 64-bit arithmetic does not apply to division. Calling this function (versus attempting an in-line implementation) is reasonably efficient and you needn't heed the warning given earlier.

Also note that this function does not automatically compute the remainder (as the x86 IDIV instruction does). The math module provides the *math.modq* function if you need to compute the remainder of the division of two 64-bit values.

HLA high-level calling sequence example:

```
// Compute Quo64 := i64 idiv j64:

math.idivq( i64, j64, Quo64 );
```

HLA low-level calling sequence example:

```
// Compute Quo64 := i64 idiv j64:

push((type dword i64[4]));
push((type dword i64));
push((type dword j64[4]));
push((type dword j64));
lea( eax, Quo64 );
push( eax );
```

```
call math.idivq;
```

```
math.modq( left:qword; right:qword; var dest:qword );
```

This routine divides one unsigned 64-bit value by another and stores the remainder into a destination variable. It computes the following:

```
dest := left & right; // Unsigned modulo
```

Since the 80x86 flags don't contain useful values after the execution of the DIV instruction, these routines also leave the flags scrambled and you can't count on flag values upon return. This routines will raise an *ex.DivideError* exception if you attempt a division by zero.

Note that the discussion of the triviality of 64-bit arithmetic does not apply to division. Calling this function (versus attempting an in-line implementation) is reasonably efficient and you needn't heed the warning given earlier.

HLA high-level calling sequence example:

```
// Compute Rem64 := i64 % j64:

math.modq( i64, j64, Rem64 );
```

HLA low-level calling sequence example:

```
// Compute Rem64 := i64 % j64:

push((type dword i64[4]));
push((type dword i64));
push((type dword j64[4]));
push((type dword j64));
lea( eax, Rem64 );
push( eax );
call math.modq;
```

```
math.imodq( left:qword; right:qword; var dest:qword );
```

This routine divides one signed 64-bit value by another and stores the remainder into a destination variable. It computes the following:

```
dest := left % right; // Signed modulo
```

Since the 80x86 flags don't contain useful values after the execution of the IDIV instruction, these routines also leave the flags scrambled and you can't count on flag values upon return. This routines will raise an *ex.DivideError* exception if you attempt a division by zero.

Note that the discussion of the triviality of 64-bit arithmetic does not apply to division. Calling this function (versus attempting an in-line implementation) is reasonably efficient and you needn't heed the warning given earlier.

HLA high-level calling sequence example:

```
// Compute Rem64 := i64 % j64:

math.imodq( i64, j64, Rem64 );
```

HLA low-level calling sequence example:

```
// Compute Rem64 := i64 % j64:
```

```
push((type dword i64[4]));
push((type dword i64));
push((type dword j64[4]));
push((type dword j64));
lea( eax, Rem64 );
push( eax );
call math.imodq;
```

**math.mulq( left:qword; right:qword; var dest:qword );**

This routine multiplies one unsigned 64-bit value by another and stores the product into a destination variable. It computes the following:

```
dest := left * right; // Unsigned multiplication
```

This function sets the carry and overflow flags if there was an unsigned overflow during the operation.

HLA high-level calling sequence example:

```
// Compute Prod64 := i64 * j64:

math.mulq( i64, j64, Prod64 );
```

HLA low-level calling sequence example:

```
// Compute Prod64 := i64 % j64:

push((type dword i64[4]));
push((type dword i64));
push((type dword j64[4]));
push((type dword j64));
lea( eax, Prod64 );
push( eax );
call math.mulq;
```

**math.imulq( left:qword; right:qword; var dest:qword );**

This routine multiplies one signed 64-bit value by another and stores the signed product into a destination variable. It computes the following:

```
dest := left * right; // Signed multiplication
```

This function sets the carry and overflow flags if there was an unsigned overflow during the operation.

HLA high-level calling sequence example:

```
// Compute Prod64 := i64 * j64:

math.imulq( i64, j64, Prod64 );
```

HLA low-level calling sequence example:

```
// Compute Prod64 := i64 % j64:

push((type dword i64[4]));
push((type dword i64));
push((type dword j64[4]));
push((type dword j64));
lea( eax, Prod64 );
push( eax );
call math.imulq;
```

**math.negq( source:qword; var dest:qword );**

This function negates (two's complement) the source operand and stores the result into the destination operand. It computes the following:

```
dest := -source;
```

This function leaves the 80x86 flags containing the same values one would expect after the execution of the NEG instruction

HLA high-level calling sequence example:

```
// Compute Neg64 := -i64:

math.negq( i64, Neg64 );
```

HLA low-level calling sequence example:

```
// Compute Neg64 := -i64:

push((type dword i64[4]));
push((type dword i64));
lea( eax, Neg64 );
push( eax );
call math.negq;
```

**math.andq( left:qword; right:qword; var dest:qword );**

This routine logically ANDs two quad-word 64-bit integer values. It computes the following:

```
dest := left & right; // "&" implies bitwise AND operation
```

This routine sets the 80x86 flags exactly the same way that the standard AND instruction does. In particular, you may test the zero flag afterwards for a zero result, you can test the sign flag to determine if there was a negative result, the carry and overflow flags are both clear.

HLA high-level calling sequence example:

```
// Compute Dest64 := i64 & j64:

math.andq( i64, j64, Dest64 );
```

HLA low-level calling sequence example:

```
// Compute Dest64 := i64 & j64:
```



```

push((type dword i64[4]));
push((type dword i64));
push((type dword j64[4]));
push((type dword j64));
lea( eax, Dest64 );
push( eax );
call math.andq;

```

**math.orq( left:qword; right:qword; var dest:qword );**

This function logically ORs two 64-bit values and stores the result into a 64-bit destination variable. It computes the following:

```
dest := left | right;
```

These routines set the 80x86 flags exactly the same way that the standard OR instruction does. In particular, you may test the zero flag afterwards for a zero result, you can test the sign flag to determine if there was a negative result, the carry and overflow flags are both clear.

HLA high-level calling sequence example:

```

// Compute Dest64 := i64 | j64:

math.orq( i64, j64, Dest64 );

```

HLA low-level calling sequence example:

```

// Compute Dest64 := i64 | j64:

push((type dword i64[4]));
push((type dword i64));
push((type dword j64[4]));
push((type dword j64));
lea( eax, Dest64 );
push( eax );
call math.orq;

```

**math.xorq( left:qword; right:qword; var dest:qword );**

This function logically XORs two 64-values and stores the result into a 64-bit variable. It computes the following:

```
dest := left ^ right;
```

This function sets the 80x86 flags exactly the same way that the standard XOR instruction does. In particular, you may test the zero flag afterwards for a zero result, you can test the sign flag to determine if there was a negative result, the carry and overflow flags are both clear.

HLA high-level calling sequence example:

```

// Compute Dest64 := i64 ^ j64:

math.xorq( i64, j64, Dest64 );

```

HLA low-level calling sequence example:

```
// Compute Dest64 := i64 ^ j64:
```

```

push((type dword i64[4]));
push((type dword i64));
push((type dword j64[4]));
push((type dword j64));
lea( eax, Dest64 );
push( eax );
call math.xorq;

```

**math.notq( source:qword; var dest:qword );**

This function inverts all the bits in the source operand and stores the result into the destination operand. It computes the following:

```
dest := ~source;
```

This function leaves the 80x86 flags containing the same values one would expect after the execution of the NOT instruction. Extended precision NOT is an especially trivial operation to compute manually; you should carefully consider whether you really want to use this function. Consistent flag results is probably the only good reason for using this function.

HLA high-level calling sequence example:

```

// Compute Neg64 := not(i64):

math.notq( i64, Not64 );

```

HLA low-level calling sequence example:

```

// Compute Neg64 := not(i64):

push((type dword i64[4]));
push((type dword i64));
lea( eax, Not64 );
push( eax );
call math.notq;

```

**math.shlq( count:uns32; source:qword; var dest:qword );**

This function logically shifts left a 64-bit value the number of bits specified by the *count* operand. It stores the result into the 64-bit dest operand. It computes the following:

```
dest := source << count; // Logical shift left operation
```

These routines set the 80x86 flags exactly the same way that the standard SHL instruction does. In particular, you may test the zero flag afterwards for a zero result, you can test the sign flag to determine if there was a negative result, the carry flag contains the last carry out of the H.O. bit, and the overflow flag is set if the last shift caused a sign change.

HLA high-level calling sequence example:

```

// Compute Dest64 := j64 << i32:

math.shlq( i32, j64, Dest64 );

```

HLA low-level calling sequence example:

```
// Compute Dest64 := j64 << i32:

push( i32 );
push((type dword j64[4]));
push((type dword j64));
lea( eax, Dest64 );
push( eax );
call math.shlq;
```

**math.shrq( count:uns32; source:qword; var dest:qword );**

This function logically shifts right a 64-bit value the number of bits specified by the *count* operand. It stores the result into the 64-bit dest operand. It computes the following:

```
dest := source >> count; // Logical shift right operation
```

These routines set the 80x86 flags exactly the same way that the standard SHR instruction does. In particular, you may test the zero flag afterwards for a zero result, you can test the sign flag to determine if there was a negative result, the carry flag contains the last carry out of the H.O. bit, and the overflow flag is set if the last shift caused a sign change.

HLA high-level calling sequence example:

```
// Compute Dest64 := j64 >> i32:

math.shrq( i32, j64, Dest64 );
```

HLA low-level calling sequence example:

```
// Compute Dest64 := j64 >> i32:

push( i32 );
push((type dword j64[4]));
push((type dword j64));
lea( eax, Dest64 );
push( eax );
call math.shrq;
```

## 21.4 128-Bit Arithmetic and Logical Operations

The HLA Standard Library provides a complete set of arithmetic and logical operations for 128-bit integers. Extended precision arithmetic is fairly straight forward and an in-line coding of some of these functions will generally be faster than calling these functions

Another reason (beyond the procedure call overhead) that these procedures are slower than the in-line code is that the standard extended precision add sequence does not set the zero flag properly; these procedures have to execute several additional instructions to preserve the carry, sign, and overflow flags as well as properly set the zero flag. So, for example, if you don't use the value of the zero flag upon return, all this extra work goes to waste.

These procedures are convenient to use and are perfectly acceptable when performance is not an issue. Another advantage is that these routines work memory to memory and don't disturb the values in any registers; and also, these routines use a "three-address" form that allows a different destination address (i.e., the destination does not have to be the same as one of the source operands. Of course, as already mentioned, these functions tend to set the flags the same way the basic machine instructions would set them, a big advantage if you are testing the flags after extended-precision arithmetic operations.

```
math.addl( left:lword; right:lword; var dest:lword );
```

This routine adds two 128-bit integer values and stores the result in a 128-bit destination memory location. The values may be signed or unsigned. This routine computes the following:

```
dest := left + right;
```

This function sets the 80x86 flags exactly the same way that the standard ADD instruction does. In particular, you may test the zero flag afterwards for a zero result, you can test the sign flag to determine if there was a negative result, the carry and overflow flags denote unsigned or signed overflow (respectively).

HLA high-level calling sequence example:

```
// Compute Sum128 := i128 + j128:

math.addl( i128, j128, Sum128 );
```

HLA low-level calling sequence example:

```
// Compute Sum128 := i128 + j128:

push((type dword i128[12]));
push((type dword i128[8]));
push((type dword i128[4]));
push((type dword i128));
push((type dword j128[12]));
push((type dword j128[8]));
push((type dword j128[4]));
push((type dword j128));
lea( eax, Sum128 );
push( eax );
call math.addl;
```

```
math.subl( left:lword; right:lword; var dest:lword );
```

This function subtracts two 128-bit integer values and stores their difference in *dest*. The values may be signed or unsigned. This function computes the following:

```
dest := left - right;
```

This function sets the 80x86 flags exactly the same way that the standard SUB instruction does. In particular, you may test the zero flag afterwards for a zero result, you can test the sign flag to determine if there was a negative result, the carry and overflow flags denote unsigned or signed overflow (respectively).

HLA high-level calling sequence example:

```
// Compute Dif128 := i128 - j128:

math.subl( i128, j128, Dif128 );
```

HLA low-level calling sequence example:

```
// Compute Dif128 := i128 - j128:

push((type dword i128[12]));
push((type dword i128[8]));
push((type dword i128[4]));
```

```

push((type dword i128));
push((type dword j128[12]));
push((type dword j128[8]));
push((type dword j128[4]));
push((type dword j128));
lea( eax, Dif128 );
push( eax );
call math.subl;

```

**math.divl( left:lword; right:lword; var dest:lword );**

This routine divides one unsigned 128-bit value by another. It computes the following:

```
dest := left div right;
```

Since the 80x86 flags don't contain useful values after the execution of the DIV instruction, these routines also leave the flags scrambled and you can't count on flag values upon return. This routines will raise an *ex.DivideError* exception if you attempt a division by zero.

Note that the discussion of the triviality of 128-bit arithmetic does not apply to division. Calling this function (versus attempting an in-line implementation) is reasonably efficient and you needn't heed the warning given earlier.

Also note that this function does not automatically compute the remainder (as the x86 DIV instruction does). The math module provides the *math.modl* function if you need to compute the remainder of the division of two 128-bit values.

HLA high-level calling sequence example:

```

// Compute Quo128 := i128 div j128:

math.divl( i128, j128, Quo128 );

```

HLA low-level calling sequence example:

```

// Compute Quo128 := i128 div j128:

push((type dword i128[12]));
push((type dword i128[8]));
push((type dword i128[4]));
push((type dword i128));
push((type dword j128[12]));
push((type dword j128[8]));
push((type dword j128[4]));
push((type dword j128));
lea( eax, Quo128 );
push( eax );
call math.divl;

```

**math.idivl( left:lword; right:lword; var dest:lword );**

This routine divides one signed 128-bit value by another. It computes the following:

```
dest := left idiv right;
```

Since the 80x86 flags don't contain useful values after the execution of the IDIV instruction, these routines also leave the flags scrambled and you can't count on flag values upon return. This routines will raise an *ex.DivideError* exception if you attempt a division by zero.

Note that the discussion of the triviality of 128-bit arithmetic does not apply to division. Calling this function (versus attempting an in-line implementation) is reasonably efficient and you needn't heed the warning given earlier.

Also note that this function does not automatically compute the remainder (as the x86 IDIV instruction does). The math module provides the *math.modl* function if you need to compute the remainder of the division of two 128-bit values.

HLA high-level calling sequence example:

```
// Compute Quo128 := i128 idiv j128:

math.idivl( i128, j128, Quo128 );
```

HLA low-level calling sequence example:

```
// Compute Quo128 := i128 idiv j128:

push((type dword i128[12]));
push((type dword i128[8]));
push((type dword i128[4]));
push((type dword i128));
push((type dword j128[12]));
push((type dword j128[8]));
push((type dword j128[4]));
push((type dword j128));
lea( eax, Quo128 );
push( eax );
call math.idivl;
```

**math.modl( left:lword; right:lword; var dest:lword );**

This routine divides one unsigned 128-bit value by another and stores the remainder into a destination variable. It computes the following:

```
dest := left & right; // Unsigned modulo
```

Since the 80x86 flags don't contain useful values after the execution of the DIV instruction, these routines also leave the flags scrambled and you can't count on flag values upon return. This routines will raise an *ex.DivideError* exception if you attempt a division by zero.

Note that the discussion of the triviality of 128-bit arithmetic does not apply to division. Calling this function (versus attempting an in-line implementation) is reasonably efficient and you needn't heed the warning given earlier.

HLA high-level calling sequence example:

```
// Compute Rem128 := i128 % j128:

math.modl( i128, j128, Rem128 );
```

HLA low-level calling sequence example:

```
// Compute Rem128 := i128 % j128:

push((type dword i128[12]));
push((type dword i128[8]));
push((type dword i128[4]));
push((type dword i128));
```

```

push((type dword j128[12]));
push((type dword j128[8]));
push((type dword j128[4]));
push((type dword j128));
lea( eax, Rem128 );
push( eax );
call math.modl;

```

**math.imodl( left:lword; right:lword; var dest:lword );**

This routine divides one signed 128-bit value by another and stores the remainder into a destination variable. It computes the following:

```
dest := left % right; // Signed modulo
```

Since the 80x86 flags don't contain useful values after the execution of the IDIV instruction, these routines also leave the flags scrambled and you can't count on flag values upon return. This routines will raise an *ex.DivideError* exception if you attempt a division by zero.

Note that the discussion of the triviality of 128-bit arithmetic does not apply to division. Calling this function (versus attempting an in-line implementation) is reasonably efficient and you needn't heed the warning given earlier.

HLA high-level calling sequence example:

```

// Compute Rem128 := i128 % j128:

math.imodl( i128, j128, Rem128 );

```

HLA low-level calling sequence example:

```

// Compute Rem128 := i128 % j128:

push((type dword i128[12]));
push((type dword i128[8]));
push((type dword i128[4]));
push((type dword i128));
push((type dword j128[12]));
push((type dword j128[8]));
push((type dword j128[4]));
push((type dword j128));
lea( eax, Rem128 );
push( eax );
call math.imodl;

```

**math.mull( left:lword; right:lword; var dest:lword );**

This routine multiplies one unsigned 128-bit value by another and stores the product into a destination variable. It computes the following:

```
dest := left * right; // Unsigned multiplication
```

This function sets the carry and overflow flags if there was an unsigned overflow during the operation.

HLA high-level calling sequence example:

```
// Compute Prod128 := i128 * j128:
```

```
math.mull( i128, j128, Prod128 );
```

HLA low-level calling sequence example:

```
// Compute Prod128 := i128 % j128:

push((type dword i128[12]));
push((type dword i128[8]));
push((type dword i128[4]));
push((type dword i128));
push((type dword j128[12]));
push((type dword j128[8]));
push((type dword j128[4]));
push((type dword j128));
lea( eax, Prod128 );
push( eax );
call math.mull;
```

```
math.imull( left:lword; right:lword; var dest:lword );
```

This routine multiplies one signed 128-bit value by another and stores the signed product into a destination variable. It computes the following:

```
dest := left * right; // Signed multiplication
```

This function sets the carry and overflow flags if there was an unsigned overflow during the operation.

HLA high-level calling sequence example:

```
// Compute Prod128 := i128 * j128:

math.imull( i128, j128, Prod128 );
```

HLA low-level calling sequence example:

```
// Compute Prod128 := i128 % j128:

push((type dword i128[12]));
push((type dword i128[8]));
push((type dword i128[4]));
push((type dword i128));
push((type dword j128[12]));
push((type dword j128[8]));
push((type dword j128[4]));
push((type dword j128));
lea( eax, Prod128 );
push( eax );
call math.imull;
```

```
math.negl( source:lword; var dest:lword );
```

This function negates (two's complement) the source operand and stores the result into the destination operand. It computes the following:

```
dest := -source;
```



This function leaves the 80x86 flags containing the same values one would expect after the execution of the NEG instruction

HLA high-level calling sequence example:

```
// Compute Neg128 := -i128:

math.neg1( i128, Neg128 );
```

HLA low-level calling sequence example:

```
// Compute Neg128 := -i128:

push((type dword i128[12]));
push((type dword i128[8]));
push((type dword i128[4]));
push((type dword i128));
lea( eax, Neg128 );
push( eax );
call math.neg1;
```

**math.and1( left:lword; right:lword; var dest:lword );**

This routine logically ANDs two 128-bit values. It computes the following:

```
dest := left & right; // "&" implies bitwise AND operation
```

This routine sets the 80x86 flags exactly the same way that the standard AND instruction does. In particular, you may test the zero flag afterwards for a zero result, you can test the sign flag to determine if there was a negative result, the carry and overflow flags are both clear.

HLA high-level calling sequence example:

```
// Compute Dest128 := i128 & j128:

math.and1( i128, j128, Dest128 );
```

HLA low-level calling sequence example:

```
// Compute Dest128 := i128 & j128:

push((type dword i128[12]));
push((type dword i128[8]));
push((type dword i128[4]));
push((type dword i128));
push((type dword j128[12]));
push((type dword j128[8]));
push((type dword j128[4]));
push((type dword j128));
lea( eax, Dest128 );
push( eax );
call math.and1;
```

**math.orl( left:lword; right:lword; var dest:lword );**

This function logically ORs two 128-bit values and stores the result into a 128-bit destination variable. It computes the following:

```
dest := left | right; // "|" implies bitwise logical OR
```

This function sets the 80x86 flags exactly the same way that the standard OR instruction does. In particular, you may test the zero flag afterwards for a zero result, you can test the sign flag to determine if there was a negative result, the carry and overflow flags are both clear.

HLA high-level calling sequence example:

```
// Compute Dest128 := i128 | j128:

math.orl( i128, j128, Dest128 );
```

HLA low-level calling sequence example:

```
// Compute Dest128 := i128 | j128:

push((type dword i128[12]));
push((type dword i128[8]));
push((type dword i128[4]));
push((type dword i128));
push((type dword j128[12]));
push((type dword j128[8]));
push((type dword j128[4]));
push((type dword j128));
lea( eax, Dest128 );
push( eax );
call math.orl;
```

**math.xorl( left:lword; right:lword; var dest:lword );**

This function logically XORs two 128-values and stores the result into a 128-bit variable. It computes the following:

```
dest := left ^ right; // "^" denotes bitwise exclusive-OR.
```

This function sets the 80x86 flags exactly the same way that the standard XOR instruction does. In particular, you may test the zero flag afterwards for a zero result, you can test the sign flag to determine if there was a negative result, the carry and overflow flags are both clear.

HLA high-level calling sequence example:

```
// Compute Dest128 := i128 ^ j128:

math.xorl( i128, j128, Dest128 );
```

HLA low-level calling sequence example:

```
// Compute Dest128 := i128 ^ j128:

push((type dword i128[12]));
push((type dword i128[8]));
push((type dword i128[4]));
```

```

push((type dword i128));
push((type dword j128[12]));
push((type dword j128[8]));
push((type dword j128[4]));
push((type dword j128));
lea( eax, Dest128 );
push( eax );
call math.xorl;

```

**math.notl( source:lword; var dest:lword );**

This function inverts all the bits in the source operand and stores the result into the destination operand. It computes the following:

```
dest := ~source;
```

This function leaves the 80x86 flags containing the same values one would expect after the execution of the NOT instruction. Extended precision NOT is an especially trivial operation to compute manually; you should carefully consider whether you really want to use this function. Consistent flag results is probably the only good reason for using this function.

HLA high-level calling sequence example:

```

// Compute Neg128 := not(i128):

math.notl( i128, Not128 );

```

HLA low-level calling sequence example:

```

// Compute Neg128 := not(i128):

push((type dword i128[12]));
push((type dword i128[8]));
push((type dword i128[4]));
push((type dword i128));
lea( eax, Not128 );
push( eax );
call math.notl;

```

**math.shll( count:uns32; source:lword; var dest:lword );**

This function logically shifts left a 128-bit value the number of bits specified by the *count* operand. It stores the result into the 128-bit dest operand. It computes the following:

```
dest := source << count; // Logical shift left operation
```

This function sets the 80x86 flags exactly the same way that the standard SHL instruction does. In particular, you may test the zero flag afterwards for a zero result, you can test the sign flag to determine if there was a negative result, the carry flag contains the last carry out of the H.O. bit, and the overflow flag is set if the last shift caused a sign change.

HLA high-level calling sequence example:

```

// Compute Dest128 := j128 << i32:

math.shll( i32, j128, Dest128 );

```

HLA low-level calling sequence example:

```
// Compute Dest128 := j128 << i32:

push( i32 );
push((type dword j128[12]));
push((type dword j128[8]));
push((type dword j128[4]));
push((type dword j128));
lea( eax, Dest128 );
push( eax );
call math.shll;
```

**math.shrl( count:uns32; source:lword; var dest:lword );**

This function logically shifts right a 128-bit value the number of bits specified by the *count* operand. It stores the result into the 128-bit dest operand. It computes the following:

```
dest := source >> count; // Logical shift right operation
```

This function sets the 80x86 flags exactly the same way that the standard SHR instruction does. In particular, you may test the zero flag afterwards for a zero result, you can test the sign flag to determine if there was a negative result, the carry flag contains the last carry out of the H.O. bit, and the overflow flag is set if the last shift caused a sign change.

HLA high-level calling sequence example:

```
// Compute Dest128 := j128 >> i32:

math.shrl( i32, j128, Dest128 );
```

HLA low-level calling sequence example:

```
// Compute Dest128 := j128 >> i32:

push( i32 );
push((type dword j128[12]));
push((type dword j128[8]));
push((type dword j128[4]));
push((type dword j128));
lea( eax, Dest128 );
push( eax );
call math.shrl;
```

## 21.5 Transcendental, Logarithmic, and Other Floating-Point Operations

The HLA Standard Library contains a wide variety of transcendental and logarithmic functions. All of these instructions use the FPU for their computations, they do not use SSE-type floating-point instructions. These functions all assume that the CPU is in floating-point mode (that is, you've not executed any MMX instructions in your program or you haven't executed any MMX instructions since you last executed an EMMS instruction) and that the FPU stack is valid.

Because the x86 FPU supports three different real data types, each of the functions in the HLA Standard Library Math module provide variants that work on single-precision (32-bit), double-precision (64-bit), and extended-precision (80-bit) memory operands. The library also includes a version of each function that operates

on the (80-bit) value currently on the FPU's top of stack. If "*fcn*" represents a specific mathematical function name, then the HLA Standard Library generally provides the following four actual functions:

```
fcn32( r32:real32 );// Expects a real32 operand
fcn64( r64:real64 );// Expects a real64 operand
fcn80( r80:real80 );// Expects a real80 operand
_fcn();                // Expects a real80 operand on the FPU top of stack
```

In addition to these four functions, the standard library will also provide a macro, simply named "*fcn*", that overloads these four functions and will automatically select the appropriate function to call based on the number of operands (zero or one) and the type of the operand (real32, real64, or real80).

Some of the functions in the Math module mirror x86 FPU instructions. The purpose of such functions is to handle range reduction and other operations needed to guarantee a correct or most precise result.

All of these functions leave an 80-bit result sitting on the top of the floating-point stack (except the *math.sincos* function, which leaves two values sitting on the FPU stack). In general, you should not count on any more significant bits than the number of bits in the original operand. That is, if you pass a 32-bit or 64-bit value to one of these functions, then you should save the result in a like-sized destination variable. Arithmetic precision is only as good as the original operand(s), so avoid false precision and store the results in appropriately-sized destination variables.

To save space, this document describes each class of functions that compute the same transcendental/logarithmic value (except for size) in a single section.

```
#macro math.sin; @returns( "st0" );// Overloads the following functions:
procedure math._sin; @returns( "st0" );
procedure math.sin32( r32: real32 ); @returns( "st0" );
procedure math.sin64( r64: real64 ); @returns( "st0" );
procedure math.sin80( r80: real80 ); @returns( "st0" );
```

These five functions compute the sine of their parameter value. The parameter value must specify an angle in radians.

The *math.sin* function is actually a macro that overloads the remaining four functions. If a *math.sin* invocation doesn't contain any parameters, then this macro expands to a call to the *math.\_sin* function which computes the sine of the value on the FPU stack (ST0). With a single parameter of the appropriate type, the macro expands to one of the other three functions with the appropriate parameter type.

The *math.\_sin()* call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the CPU stack using the standard HLA parameter passing mechanism.

The purpose of this function (which has a corresponding FPU instruction) is to get out-of-range values into the legal range before computing the sine via the FPU FSIN instruction.

HLA high-level calling sequence examples:

```
// Compute y := sin(x):

math.sin32( x32 );
fstp( y32 );

math.sin64( x64 );
fstp( y64 );

math.sin80( x80 );
fstp( y80 );

fld( x80 );
math._sin();
fstp( y80 );

// Using the math.sin macro:
```

```

math.sin( x32 );
fstp( y32 );

math.sin( x64 );
fstp( y64 );

math.sin( x80 );
fstp( y80 );

fld( x80 );
math.sin();
fstp( y80 );

```

HLA low-level calling sequence example:

```

// Compute y := sin(x):

push( x32 );
call math.sin32;
fstp( y32 );

push( (type dword x64[4]) );
push( (type dword x64[0]) );
call math.sin64;
fstp( y64 );

pushw( 0 );           // Must dword align operand
push( (type word x80[8]) );
push( (type dword x80[4]) );
push( (type dword x80[0]) );
call math.sin80;
fstp( y80 );

fld( x80 );
call math.sin;
fstp( y80 );

```

```

#macro math.cos; @returns( "st0" ); // Overloads the following functions:
procedure math._cos; @returns( "st0" );
procedure math.cos32( r32: real32 ); @returns( "st0" );
procedure math.cos64( r64: real64 ); @returns( "st0" );
procedure math.cos80( r80: real80 ); @returns( "st0" );

```

These five functions compute the cosine of their parameter value. The parameter value must specify an angle in radians.

The *math.cos* function is actually a macro that overloads the remaining four functions. If a *math.cos* invocation doesn't contain any parameters, then this macro expands to a call to the *math.\_cos* function which computes the cosine of the value on the FPU stack (ST0). With a single parameter of the appropriate type, the macro expands to one of the other three functions with the appropriate parameter type.

The *math.cos()* call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the CPU stack using the standard HLA parameter passing mechanism.

The purpose of this function (which has a corresponding FPU instruction) is to get out-of-range values into the legal range before computing the cosine via the FPU FCOS instruction.

HLA high-level calling sequence examples:

```
// Compute y := cos(x):

math.cos32( x32 );
fstp( y32 );

math.cos64( x64 );
fstp( y64 );

math.cos80( x80 );
fstp( y80 );

fld( x80 );
math._cos();
fstp( y80 );

// Ucosg the math.cos macro:

math.cos( x32 );
fstp( y32 );

math.cos( x64 );
fstp( y64 );

math.cos( x80 );
fstp( y80 );

fld( x80 );
math.cos();
fstp( y80 );
```

HLA low-level calling sequence example:

```
// Compute y := cos(x):

push( x32 );
call math.cos32;
fstp( y32 );

push( (type dword x64[4]) );
push( (type dword x64[0]) );
call math.cos64;
fstp( y64 );

pushw( 0 );           // Must dword align operand
push( (type word x80[8]) );
push( (type dword x80[4]) );
push( (type dword x80[0]) );
call math.cos80;
fstp( y80 );

fld( x80 );
call math.cos;
fstp( y80 );
```

```
#macro math.tan; @returns( "st0" ); // Overloads the following functions:
procedure math._tan; @returns( "st0" );
procedure math.tan32( r32: real32 ); @returns( "st0" );
procedure math.tan64( r64: real64 ); @returns( "st0" );
procedure math.tan80( r80: real80 ); @returns( "st0" );
```

These five functions compute the tangent of their parameter value. The parameter value must specify an angle in radians.

The *math.tan* function is actually a macro that overloads the remaining four functions. If a *math.tan* invocation doesn't contain any parameters, then this macro expands to a call to the *math.\_tan* function which computes the tangent of the value on the FPU stack (ST0). With a single parameter of the appropriate type, the macro expands to one of the other three functions with the appropriate parameter type.

The *math.\_tan()* call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the CPU stack using the standard HLA parameter passing mechanism.

The purpose of this function (which has a corresponding FPU instruction) is to get out-of-range values into the legal range before computing the cosine via the FPU FTAN instruction.

HLA high-level calling sequence examples:

```
// Compute y := tan(x):

math.tan32( x32 );
fstp( y32 );

math.tan64( x64 );
fstp( y64 );

math.tan80( x80 );
fstp( y80 );

fld( x80 );
math._tan();
fstp( y80 );

// Utang the math.tan macro:

math.tan( x32 );
fstp( y32 );

math.tan( x64 );
fstp( y64 );

math.tan( x80 );
fstp( y80 );

fld( x80 );
math.tan();
fstp( y80 );
```

HLA low-level calling sequence example:

```
// Compute y := tan(x):

push( x32 );
call math.tan32;
fstp( y32 );

push( (type dword x64[4]) );
```



```

push( (type dword x64[0]) );
call math.tan64;
fstp( y64 );

pushw( 0 );           // Must dword align operand
push( (type word x80[8]) );
push( (type dword x80[4]) );
push( (type dword x80[0]) );
call math.tan80;
fstp( y80 );

fld( x80 );
call math.tan;
fstp( y80 );

#macro math.sincos; // Overloads the following functions:
procedure math._sincos;
procedure math.sincos32( r32: real32 );
procedure math.sincos64( r64: real64 );
procedure math.sincos80( r80: real80 );

```

These five functions compute the sine and cosine of their parameter value. The parameter value must specify an angle in radians.

The *math.sincos* function is actually a macro that overloads the remaining four functions. If a *math.sincos* invocation doesn't contain any parameters, then this macro expands to a call to the *math.\_sincos* function which computes the sine and cosine of the value on the FPU stack (ST0). With a single parameter of the appropriate type, the macro expands to one of the other three functions with the appropriate parameter type.

The *math.\_sincos()* call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the CPU stack using the standard HLA parameter passing mechanism.

The purpose of this function (which has a corresponding FPU instruction) is to get out-of-range values into the legal range before computing the sine and cosine via the FPU FSINCOS instruction.

This function computes two return results (the sine and the cosine) and leaves the two values on the FPU stack at ST0 and ST1.

HLA high-level calling sequence examples:

```

// Compute y := sincos(x):

math.sincos32( x32 );
fstp( y32 );

math.sincos64( x64 );
fstp( y64 );

math.sincos80( x80 );
fstp( y80 );

fld( x80 );
math._sincos();
fstp( y80 );

// Usincosg the math.sincos macro:

math.sincos( x32 );
fstp( y32 );

math.sincos( x64 );

```

```

fstp( y64 );

math.sincos( x80 );
fstp( y80 );

fld( x80 );
math.sincos();
fstp( y80 );

```

HLA low-level calling sequence example:

```

// Compute y := sincos(x):

push( x32 );
call math.sincos32;
fstp( y32 );

push( (type dword x64[4]) );
push( (type dword x64[0]) );
call math.sincos64;
fstp( y64 );

pushw( 0 );           // Must dword align operand
push( (type word x80[8]) );
push( (type dword x80[4]) );
push( (type dword x80[0]) );
call math.sincos80;
fstp( y80 );

fld( x80 );
call math.sincos;
fstp( y80 );

```

```

#macro math.atan; @returns( "st0" ); // Overloads the following functions:
procedure math._atan; @returns( "st0" );

procedure math.atan32( r32: real32 ); @returns( "st0" );
procedure math.atan64( r64: real64 ); @returns( "st0" );
procedure math.atan80( r80: real80 ); @returns( "st0" );

```

These five functions compute the arc tangent of their parameter value. The parameter value must specify an angle in radians.

The *math.atan* function is actually a macro that overloads the remaining four functions. If a *math.atan* invocation doesn't contain any parameters, then this macro expands to a call to the *math.\_atan* function which computes the arc tangent of the value on the FPU stack (ST0). With a single parameter of the appropriate type, the macro expands to one of the other three functions with the appropriate parameter type.

The *math.\_atan()* call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the CPU stack using the standard HLA parameter passing mechanism.

The purpose of this function (which has a corresponding FPU instruction) is to get out-of-range values into the legal range before computing the cosine via the FPU FATAN instruction.

HLA high-level calling sequence examples:

```

// Compute y := atan(x):

math.atan32( x32 );

```

```

fstp( y32 );

math.atan64( x64 );
fstp( y64 );

math.atan80( x80 );
fstp( y80 );

fld( x80 );
math._atan();
fstp( y80 );

// Uatang the math.atan macro:

math.atan( x32 );
fstp( y32 );

math.atan( x64 );
fstp( y64 );

math.atan( x80 );
fstp( y80 );

fld( x80 );
math.atan();
fstp( y80 );

```

HLA low-level calling sequence example:

```

// Compute y := atan(x):

push( x32 );
call math.atan32;
fstp( y32 );

push( (type dword x64[4]) );
push( (type dword x64[0]) );
call math.atan64;
fstp( y64 );

pushw( 0 );           // Must dword align operand
push( (type word x80[8]) );
push( (type dword x80[4]) );
push( (type dword x80[0]) );
call math.atan80;
fstp( y80 );

fld( x80 );
call math.atan;
fstp( y80 );

```

```

#macro math.cot; @returns( "st0" ); // Overloads the following functions:
procedure math._cot; @returns( "st0" );
procedure math.cot32( r32: real32 ); @returns( "st0" );

```

```

procedure math.cot64( r64: real64 ); @returns( "st0" );
procedure math.cot80( r80: real80 ); @returns( "st0" );

```

These five functions compute the cotangent (1/tan) of their parameter value. The parameter value must specify an angle in radians.

The *math.cot* function is actually a macro that overloads the remaining four functions. If a *math.cot* invocation doesn't contain any parameters, then this macro expands to a call to the *math.\_cot* function which computes the cotangent of the value on the FPU stack (ST0). With a single parameter of the appropriate type, the macro expands to one of the other three functions with the appropriate parameter type.

The *math.\_cot()* call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the CPU stack using the standard HLA parameter passing mechanism.

HLA high-level calling sequence examples:

```

// Compute y := cot(x):

math.cot32( x32 );
fstp( y32 );

math.cot64( x64 );
fstp( y64 );

math.cot80( x80 );
fstp( y80 );

fld( x80 );
math._cot();
fstp( y80 );

// Using the math.cot macro:

math.cot( x32 );
fstp( y32 );

math.cot( x64 );
fstp( y64 );

math.cot( x80 );
fstp( y80 );

fld( x80 );
math.cot();
fstp( y80 );

```

HLA low-level calling sequence example:

```

// Compute y := cot(x):

push( x32 );
call math.cot32;
fstp( y32 );

push( (type dword x64[4]) );
push( (type dword x64[0]) );
call math.cot64;
fstp( y64 );

pushw( 0 );           // Must dword align operand

```

```

push( (type word x80[8]) );
push( (type dword x80[4]) );
push( (type dword x80[0]) );
call math.cot80;
fstp( y80 );

fld( x80 );
call math.cot;
fstp( y80 );

#macro math.csc // Macro that overloads the following four functions:
procedure math._csc; @returns( "st0" );
procedure math.csc32( r32:real32 ); @returns( "st0" );
procedure math.csc64( r64: real64 ); @returns( "st0" );
procedure math.csc80( r80: real80 ); @returns( "st0" );

```

These five functions compute the cosecant (1/sin) of their parameter value. The parameter value must specify an angle in radians.

The *math.csc* function is actually a macro that overloads the remaining four functions. If a *math.csc* invocation doesn't contain any parameters, then this macro expands to a call to the *math.\_csc* function which computes the cosecant of the value on the FPU stack (ST0). With a single real parameter, the macro expands to one of the other three functions with the appropriate parameter type.

The "*math.\_csc()*;" call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the stack using the standard HLA parameter passing mechanism.

HLA high-level calling sequence examples:

```

// Compute y := csc(x):

math.csc32( x32 );
fstp( y32 );

math.csc64( x64 );
fstp( y64 );

math.csc80( x80 );
fstp( y80 );

fld( x80 );
math._csc();
fstp( y80 );

// Using the math.csc macro:

math.csc( x32 );
fstp( y32 );

math.csc( x64 );
fstp( y64 );

math.csc( x80 );
fstp( y80 );

fld( x80 );
math.csc();
fstp( y80 );

```

HLA low-level calling sequence example:

```
// Compute y := csc(x):

push( x32 );
call math.csc32;
fstp( y32 );

push( (type dword x64[4]) );
push( (type dword x64[0]) );
call math.csc64;
fstp( y64 );

pushw( 0 );           // Must dword align operand
push( (type word x80[8]) );
push( (type dword x80[4]) );
push( (type dword x80[0]) );
call math.csc80;
fstp( y80 );

fld( x80 );
call math.csc;
fstp( y80 );
```

**#macro math.sec; // Macro that overloads the following four functions:**

```
procedure math._sec; @returns( "st0" );
procedure math.sec32( r32:real32 ); @returns( "st0" );
procedure math.sec64( r64: real64 ); @returns( "st0" );
procedure math.sec80( r80: real80 ); @returns( "st0" );
```

These five functions compute the secant ( $1/\cos$ ) of their parameter value. The parameter value must specify an angle in radians.

The *math.sec* function is actually a macro that overloads the remaining four functions. If a *math.sec* invocation doesn't contain any parameters, then this macro expands to a call to the *math.\_sec* function which computes the secant of the value on the FPU stack (ST0). With a single real parameter, the macro expands to one of the other three functions with the appropriate parameter type.

The "*math.\_sec()*;" call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the stack using the standard HLA parameter passing mechanism.

HLA high-level calling sequence examples:

```
// Compute y := sec(x):

math.sec32( x32 );
fstp( y32 );

math.sec64( x64 );
fstp( y64 );

math.sec80( x80 );
fstp( y80 );

fld( x80 );
math._sec();
fstp( y80 );
```

```
// Using the math.sec macro:
```

```
math.sec( x32 );
fstp( y32 );
```

```
math.sec( x64 );
fstp( y64 );
```

```
math.sec( x80 );
fstp( y80 );
```

```
fld( x80 );
math.sec();
fstp( y80 );
```

HLA low-level calling sequence example:

```
// Compute y := sec(x):
```

```
push( x32 );
call math.sec32;
fstp( y32 );
```

```
push( (type dword x64[4]) );
push( (type dword x64[0]) );
call math.sec64;
fstp( y64 );
```

```
pushw( 0 );           // Must dword align operand
push( (type word x80[8]) );
push( (type dword x80[4]) );
push( (type dword x80[0]) );
call math.sec80;
fstp( y80 );
```

```
fld( x80 );
call math.sec;
fstp( y80 );
```

**#macro math.asin // Macro that overloads the following four functions:**

```
procedure math._asin; @returns( "st0" );
procedure math.asin32( r32:real32 ); @returns( "st0" );
procedure math.asin64( r64: real64 ); @returns( "st0" );
procedure math.asin80( r80: real80 ); @returns( "st0" );
```

These five functions compute the arc sine ( $\sin^{-1}$ ) of their parameter value. They return an angle in radians.

The *math.asin* function is actually a macro that overloads the remaining four functions. If a *math.asin* invocation doesn't contain any parameters, then this macro expands to a call to the *math.\_asin* function which computes the arc sin of the value on the FPU stack (ST0). With a single real parameter, the macro expands to one of the other three functions with the appropriate parameter type.

The "*math.\_asin()*;" call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the stack using the standard HLA parameter passing mechanism.

HLA high-level calling sequence examples:

```

// Compute y := asin(x):

math.asin32( x32 );
fstp( y32 );

math.asin64( x64 );
fstp( y64 );

math.asin80( x80 );
fstp( y80 );

fld( x80 );
math._asin();
fstp( y80 );

// Using the math.asin macro:

math.asin( x32 );
fstp( y32 );

math.asin( x64 );
fstp( y64 );

math.asin( x80 );
fstp( y80 );

fld( x80 );
math.asin();
fstp( y80 );

```

HLA low-level calling sequence example:

```

// Compute y := asin(x):

push( x32 );
call math.asin32;
fstp( y32 );

push( (type dword x64[4]) );
push( (type dword x64[0]) );
call math.asin64;
fstp( y64 );

pushw( 0 );           // Must dword align operand
push( (type word x80[8]) );
push( (type dword x80[4]) );
push( (type dword x80[0]) );
call math.asin80;
fstp( y80 );

fld( x80 );
call math.asin;
fstp( y80 );

```

```

#macro math.acos // Macro to overload the following four functions:
procedure math._acos; @returns( "st0" );

```



```

procedure  math.acos32( r32:real32 ); @returns( "st0" );
procedure  math.acos64( r64: real64 );  @returns( "st0" );
procedure  math.acos80( r80: real80 );  @returns( "st0" );

```

These five functions compute the arc cosine ( $\cos^{-1}$ ) of their parameter value. They return an angle in radians.

The *math.acos* function is actually a macro that overloads the remaining four functions. If a *math.acos* invocation doesn't contain any parameters, then this macro expands to a call to the *math.\_acos* function which computes the arc cosine of the value on the FPU stack (ST0). With a single real parameter, the macro expands to one of the other three functions with the appropriate parameter type.

The "*math.\_acos()*;" call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the stack using the standard HLA parameter passing mechanism.

HLA high-level calling sequence examples:

```

// Compute y := acos(x):

math.acos32( x32 );
fstp( y32 );

math.acos64( x64 );
fstp( y64 );

math.acos80( x80 );
fstp( y80 );

fld( x80 );
math._acos();
fstp( y80 );

// Using the math.acos macro:

math.acos( x32 );
fstp( y32 );

math.acos( x64 );
fstp( y64 );

math.acos( x80 );
fstp( y80 );

fld( x80 );
math.acos();
fstp( y80 );

```

HLA low-level calling sequence example:

```

// Compute y := acos(x):

push( x32 );
call math.acos32;
fstp( y32 );

push( (type dword x64[4]) );
push( (type dword x64[0]) );
call math.acos64;
fstp( y64 );

```

```

pushw( 0 );           // Must dword align operand
push( (type word x80[8]) );
push( (type dword x80[4]) );
push( (type dword x80[0]) );
call math.acos80;
fstp( y80 );

fld( x80 );
call math.acos;
fstp( y80 );

```

**#macro math.acot** // Macro that overloads the following four functions:

```

procedure math._acot; @returns( "st0" );
procedure math.acot32( r32:real32 ); @returns( "st0" );
procedure math.acot64( r64: real64 ); @returns( "st0" );
procedure math.acot80( r80: real80 ); @returns( "st0" );

```

These five functions compute the arc cotangent ( $\cot^{-1}$ ) of their parameter value. They return an angle in radians.

The *math.acot* function is actually a macro that overloads the remaining four functions. If a *math.acot* invocation doesn't contain any parameters, then this macro expands to a call to the *math.\_acot* function which computes the arc cotangent of the value on the FPU stack (ST0). With a single real parameter, the macro expands to one of the other three functions with the appropriate parameter type.

The "*math.\_acot*();" call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the stack using the standard HLA parameter passing mechanism.

HLA high-level calling sequence examples:

```

// Compute y := acot(x):

math.acot32( x32 );
fstp( y32 );

math.acot64( x64 );
fstp( y64 );

math.acot80( x80 );
fstp( y80 );

fld( x80 );
math._acot();
fstp( y80 );

// Using the math.acot macro:

math.acot( x32 );
fstp( y32 );

math.acot( x64 );
fstp( y64 );

math.acot( x80 );
fstp( y80 );

fld( x80 );
math.acot();
fstp( y80 );

```

HLA low-level calling sequence example:

```
// Compute y := acot(x):

push( x32 );
call math.acot32;
fstp( y32 );

push( (type dword x64[4]) );
push( (type dword x64[0]) );
call math.acot64;
fstp( y64 );

pushw( 0 );           // Must dword align operand
push( (type word x80[8]) );
push( (type dword x80[4]) );
push( (type dword x80[0]) );
call math.acot80;
fstp( y80 );

fld( x80 );
call math.acot;
fstp( y80 );
```

```
#macro math.acsc// Overload macro that expands to one of the following:
procedure math._acsc; @returns( "st0" );
procedure math.acsc32( r32:real32 ); @returns( "st0" );
procedure math.acsc64( r64: real64 ); @returns( "st0" );
procedure math.acsc80( r80: real80 ); @returns( "st0" );
```

These five functions compute the arc cosecant ( $\csc^{-1}$ ) of their parameter value. They return an angle in radians.

The *math.acsc* function is actually a macro that overloads the remaining four functions. If a *math.acsc* invocation doesn't contain any parameters, then this macro expands to a call to the *math.\_acsc* function which computes the arc cosecant of the value on the FPU stack (ST0). With a single real parameter, the macro expands to one of the other three functions with the appropriate parameter type.

The "*math.\_acsc()*;" call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the stack using the standard HLA parameter passing mechanism.

HLA high-level calling sequence examples:

```
// Compute y := acsc(x):

math.acsc32( x32 );
fstp( y32 );

math.acsc64( x64 );
fstp( y64 );

math.acsc80( x80 );
fstp( y80 );

fld( x80 );
math._acsc();
fstp( y80 );
```

```
// Using the math.acsc macro:

math.acsc( x32 );
fstp( y32 );

math.acsc( x64 );
fstp( y64 );

math.acsc( x80 );
fstp( y80 );

fld( x80 );
math.acsc();
fstp( y80 );
```

HLA low-level calling sequence example:

```
// Compute y := acsc(x):

push( x32 );
call math.acsc32;
fstp( y32 );

push( (type dword x64[4]) );
push( (type dword x64[0]) );
call math.acsc64;
fstp( y64 );

pushw( 0 );           // Must dword align operand
push( (type word x80[8]) );
push( (type dword x80[4]) );
push( (type dword x80[0]) );
call math.acsc80;
fstp( y80 );

fld( x80 );
call math.acsc;
fstp( y80 );
```

```
#macro math.asec // Overloading macro that expands to one of:
procedure math._asec; @returns( "st0" );
procedure math.asec32( r32:real32 ); @returns( "st0" );
procedure math.asec64( r64: real64 ); @returns( "st0" );
procedure math.asec80( r80: real80 ); @returns( "st0" );
```

These five functions compute the arc secant ( $\sec^{-1}$ ) of their parameter value. They return an angle in radians.

The *math.asec* function is actually a macro that overloads the remaining four functions. If a *math.asec* invocation doesn't contain any parameters, then this macro expands to a call to the *math.\_asec* function which computes the arc secant of the value on the FPU stack (ST0). With a single real parameter, the macro expands to one of the other three functions with the appropriate parameter type.

The "*math.asec()*;" call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the stack using the standard HLA parameter passing mechanism.

HLA high-level calling sequence examples:

```
// Compute y := asec(x):

math.asec32( x32 );
fstp( y32 );

math.asec64( x64 );
fstp( y64 );

math.asec80( x80 );
fstp( y80 );

fld( x80 );
math._asec();
fstp( y80 );

// Using the math.asec macro:

math.asec( x32 );
fstp( y32 );

math.asec( x64 );
fstp( y64 );

math.asec( x80 );
fstp( y80 );

fld( x80 );
math.asec();
fstp( y80 );
```

HLA low-level calling sequence example:

```
// Compute y := asec(x):

push( x32 );
call math.asec32;
fstp( y32 );

push( (type dword x64[4]) );
push( (type dword x64[0]) );
call math.asec64;
fstp( y64 );

pushw( 0 );           // Must dword align operand
push( (type word x80[8]) );
push( (type dword x80[4]) );
push( (type dword x80[0]) );
call math.asec80;
fstp( y80 );

fld( x80 );
call math.asec;
fstp( y80 );
```

```
#macro math.twoToX // Macro that overloads the following functions:
procedure math._twoToX; @returns( "st0" );
procedure math.twoToX32( r32: real32 ); @returns( "st0" );
procedure math.twoToX64( r64: real64 ); @returns( "st0" );
procedure math.twoToX80( r80: real80 ); @returns( "st0" );
```

These five functions compute  $2^x$  of their parameter value (which is the value of x).

The *math.twoToX* function is actually a macro that overloads the remaining four functions. If a *math.twoToX* invocation doesn't contain any parameters, then this macro expands to a call to the *math.\_twoToX* function which computes  $2^x$  of the value on the FPU stack (ST0). With a single real parameter, the macro expands to one of the other three functions with the appropriate parameter type.

The "*math.\_twoToX*();" call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the stack using the standard HLA parameter passing mechanism.

HLA high-level calling sequence examples:

```
// Compute y := 2**x:

math.twoToX32( x32 );
fstp( y32 );

math.twoToX64( x64 );
fstp( y64 );

math.twoToX80( x80 );
fstp( y80 );

fld( x80 );
math._twoToX();
fstp( y80 );

// Using the math.twoToX macro:

math.twoToX( x32 );
fstp( y32 );

math.twoToX( x64 );
fstp( y64 );

math.twoToX( x80 );
fstp( y80 );

fld( x80 );
math.twoToX();
fstp( y80 );
```

HLA low-level calling sequence example:

```
// Compute y := twoToX(x):

push( x32 );
call math.twoToX32;
fstp( y32 );

push( (type dword x64[4]) );
push( (type dword x64[0]) );
call math.twoToX64;
fstp( y64 );
```

```

pushw( 0 );           // Must dword align operand
push( (type word x80[8]) );
push( (type dword x80[4]) );
push( (type dword x80[0]) );
call math.twoToX80;
fstp( y80 );

fld( x80 );
call math.twoToX;
fstp( y80 );

#macro math.TenToX // Overloads the following functions:
procedure math._tenToX; @returns( "st0" );
procedure math.tenToX32( r32:real32 ); @returns( "st0" );
procedure math.tenToX64( r64: real64 ); @returns( "st0" );
procedure math.tenToX80( r80: real80 ); @returns( "st0" );

```

These five functions compute  $10^x$  of their parameter value (which is the value of  $x$ ).

The *math.tenToX* function is actually a macro that overloads the remaining four functions. If a *math.tenToX* invocation doesn't contain any parameters, then this macro expands to a call to the *math.\_tenToX* function which computes the  $10^x$  of the value on the FPU stack (ST0). With a single real parameter, the macro expands to one of the other three functions with the appropriate parameter type.

The "*math.\_tenToX()*;" call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the stack using the standard HLA parameter passing mechanism.

HLA high-level calling sequence examples:

```

// Compute y := 10**x:

math.tenToX32( x32 );
fstp( y32 );

math.tenToX64( x64 );
fstp( y64 );

math.tenToX80( x80 );
fstp( y80 );

fld( x80 );
math._tenToX();
fstp( y80 );

// Using the math.tenToX macro:

math.tenToX( x32 );
fstp( y32 );

math.tenToX( x64 );
fstp( y64 );

math.tenToX( x80 );
fstp( y80 );

fld( x80 );
math.tenToX();
fstp( y80 );

```

HLA low-level calling sequence example:

```
// Compute y := tenToX(x):

push( x32 );
call math.tenToX32;
fstp( y32 );

push( (type dword x64[4]) );
push( (type dword x64[0]) );
call math.tenToX64;
fstp( y64 );

pushw( 0 );           // Must dword align operand
push( (type word x80[8]) );
push( (type dword x80[4]) );
push( (type dword x80[0]) );
call math.tenToX80;
fstp( y80 );

fld( x80 );
call math.tenToX;
fstp( y80 );
```

```
#macro math.exp // Overloads the following functions:
procedure math._exp; @returns( "st0" );
procedure math.exp32( r32:real32 ); @returns( "st0" );
procedure math.exp64( r64: real64 ); @returns( "st0" );
procedure math.exp80( r80: real80 ); @returns( "st0" );
```

These five functions compute  $e^x$  of their parameter value (which is the value of  $x$ ).

The *math.exp* function is actually a macro that overloads the remaining four functions. If a *math.exp* invocation doesn't contain any parameters, then this macro expands to a call to the *math.\_exp* function which computes  $e^x$  of the value on the FPU stack (ST0). With a single real parameter, the macro expands to one of the other three functions with the appropriate parameter type.

The "*math.\_exp()*;" call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the stack using the standard HLA parameter passing mechanism.

HLA high-level calling sequence examples:

```
// Compute y := e**x:

math.exp32( x32 );
fstp( y32 );

math.exp64( x64 );
fstp( y64 );

math.exp80( x80 );
fstp( y80 );

fld( x80 );
math._exp();
fstp( y80 );

// Using the math.exp macro:
```



```

math.exp( x32 );
fstp( y32 );

math.exp( x64 );
fstp( y64 );

math.exp( x80 );
fstp( y80 );

fld( x80 );
math.exp();
fstp( y80 );

```

HLA low-level calling sequence example:

```

// Compute y := e**x:

push( x32 );
call math.exp32;
fstp( y32 );

push( (type dword x64[4]) );
push( (type dword x64[0]) );
call math.exp64;
fstp( y64 );

pushw( 0 );           // Must dword align operand
push( (type word x80[8]) );
push( (type dword x80[4]) );
push( (type dword x80[0]) );
call math.exp80;
fstp( y80 );

fld( x80 );
call math.exp;
fstp( y80 );

```

```

#macro math.ytoX // Macro that overloads the following functions:
procedure math._ytoX;           // Y is at ST1, X is at ST0.
procedure math.yToX32( y32Var, x32Var ); @returns( "st0" );
procedure math.yToX64( y64Var, x64Var ); @returns( "st0" );
procedure math.yToX80( y80Var, x80Var ); @returns( "st0" );

```

These five functions compute  $Y^X$  of their parameter values (which are the values of  $y$  and  $x$ ).

The *math.yToX* function is actually a macro that overloads the remaining four functions. If a *math.yToX* invocation doesn't contain any parameters, then this macro expands to a call to the *math.\_ytoX* function which computes  $Y^X$  using the values on the FPU stack (ST0). With a single real parameter, the macro expands to one of the other three functions with the appropriate parameter type.

The "*math.\_ytoX()*;" call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the stack using the standard HLA parameter passing mechanism.

Because this function uses logarithms to compute its result, the  $y$  argument must be a non-negative value.

HLA high-level calling sequence examples:

```

// Compute z := y**x:

```

```

math.yToX32( y32, x32 );
fstp( z32 );

math.yToX64( y64, x64 );
fstp( z64 );

math.yToX80( y80, x80 );
fstp( z80 );

fld( y80 );
fld( x80 );
math._yToX();
fstp( z80 );

// Using the math.yToX macro:

math.yToX( y32, x32 );
fstp( z32 );

math.yToX( y64, x64 );
fstp( z64 );

math.yToX( y80, x80 );
fstp( z80 );

fld( y80 );
fld( x80 );
math.yToX();
fstp( z80 );

```

HLA low-level calling sequence example:

```

// Compute z := yToX( y, x ):

push( y32 );
push( x32 );
call math.yToX32;
fstp( z32 );

push( (type dword x64[4]) );
push( (type dword x64[0]) );
push( (type dword y64[4]) );
push( (type dword y64[0]) );
call math.yToX64;
fstp( z64 );

pushw( 0 );           // Must dword align operand
push( (type word y80[8]) );
push( (type dword y80[4]) );
push( (type dword y80[0]) );
pushw( 0 );           // Must dword align operand
push( (type word x80[8]) );
push( (type dword x80[4]) );
push( (type dword x80[0]) );
call math.yToX80;
fstp( z80 );

fld( y80 );

```

```
fld( x80 );
call math.yToX;
fstp( z80 );
```

```
#macro math.log // Overloads the following functions:
procedure math._log; @returns( "st0" );
procedure math.log32( r32: real32 ); @returns( "st0" );
procedure math.log64( r64: real64 ); @returns( "st0" );
procedure math.log80( r80: real80 ); @returns( "st0" );
```

These five functions compute  $\log_{10}(x)$  of their parameter value (which is the value of  $x$ ).

The *math.log* function is actually a macro that overloads the remaining four functions. If a *math.log* invocation doesn't contain any parameters, then this macro expands to a call to the *math.\_log* function which computes the base 10 log of the value on the FPU stack (ST0). With a single real parameter, the macro expands to one of the other three functions with the appropriate parameter type.

The "*math.\_log()*;" call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the stack using the standard HLA parameter passing mechanism.

HLA high-level calling sequence examples:

```
// Compute y := log(x):

math.log32( x32 );
fstp( y32 );

math.log64( x64 );
fstp( y64 );

math.log80( x80 );
fstp( y80 );

fld( x80 );
math._log();
fstp( y80 );

// Using the math.log macro:

math.log( x32 );
fstp( y32 );

math.log( x64 );
fstp( y64 );

math.log( x80 );
fstp( y80 );

fld( x80 );
math.log();
fstp( y80 );
```

HLA low-level calling sequence example:

```
// Compute y := log(x):

push( x32 );
call math.log32;
```

```

fstp( y32 );

push( (type dword x64[4]) );
push( (type dword x64[0]) );
call math.log64;
fstp( y64 );

pushw( 0 );           // Must dword align operand
push( (type word x80[8]) );
push( (type dword x80[4]) );
push( (type dword x80[0]) );
call math.log80;
fstp( y80 );

fld( x80 );
call math.log;
fstp( y80 );

```

```

#macro math.ln // Overloads the following functions:
procedure math._ln; @returns( "st0" );
procedure math.ln32( r32: real32 ); @returns( "st0" );
procedure math.ln64( r64: real64 ); @returns( "st0" );
procedure math.ln80( r80: real80 ); @returns( "st0" );

```

$\ln(x)$  [ $\log_e(x)$ ]

These five functions compute  $\log_e(x)$  of their parameter value (which is the value of  $x$ ).

The *math.ln* function is actually a macro that overloads the remaining four functions. If a *math.ln* invocation doesn't contain any parameters, then this macro expands to a call to the *math.\_ln* function which computes the base  $e$  log of the value on the FPU stack (ST0). With a single real parameter, the macro expands to one of the other three functions with the appropriate parameter type.

The "*math.\_ln()*;" call expects the parameter on the FPU stack, the other three forms pass their parameter by value on the stack using the standard HLA parameter passing mechanism.

HLA high-level calling sequence examples:

```

// Compute y := ln(x):

math.ln32( x32 );
fstp( y32 );

math.ln64( x64 );
fstp( y64 );

math.ln80( x80 );
fstp( y80 );

fld( x80 );
math._ln();
fstp( y80 );

// Using the math.ln macro:

math.ln( x32 );
fstp( y32 );

math.ln( x64 );
fstp( y64 );

```

```

math.ln( x80 );
fstp( y80 );

fld( x80 );
math.ln();
fstp( y80 );

```

HLA low-level calling sequence example:

```

// Compute y := ln(x):

push( x32 );
call math.ln32;
fstp( y32 );

push( (type dword x64[4]) );
push( (type dword x64[0]) );
call math.ln64;
fstp( y64 );

pushw( 0 );           // Must dword align operand
push( (type word x80[8]) );
push( (type dword x80[4]) );
push( (type dword x80[0]) );
call math.ln80;
fstp( y80 );

fld( x80 );
call math.ln;
fstp( y80 );

```



## 22 Memory-Mapped I/O (mmap.hhf)

The HLA Standard Library provides a set of routines for creating and manipulating memory-mapped files. Memory-mapped files are very efficient because they use the underlying operating systems' virtual memory subsystem for file I/O. When you open a memory-mapped file, the OS maps the entire file into the process' address space. Reading data from the file consists of nothing more than a memory access. Indeed, random file access is trivial in a memory mapped file system (you can treat the entire file as one huge array of characters from your software's point of view).

Although memory-mapped file access is very fast, the HLA Standard Library implementation does have a couple of limitations that make it unacceptable for some applications. First of all, as the operating system maps the file into your process' address space, memory-mapped files cannot exceed 2GBytes in size (in fact, the operating system might not even support files that large). Second, when processing existing files, you cannot extend the file's size. You may modify any data that already exists in the file, but you cannot append data to the end of the file. Third, when creating a new file, you must specify the size of the file when you first create it. You cannot open the file and then arbitrarily extend it during program execution as you can with a standard file.

### 22.1 MMAP Module

To call functions in the MMap module, you must include one of the following statements in your HLA application:

```
#include( "mmap.hhf" )
or
#include( "stdlib.hhf" )
```

Note: be sure to read the chapter on "Passing Parameters to Standard Library Routines" (parmpassing.rtf) before reading this chapter.

### 22.2 Class Fields

Note that the memory-mapping module in the HLA Standard Library is implemented as a class. This class (mmap) defines the following public fields:

**filePtr:dword;**

This field holds a pointer to the first byte of the file in memory. You must not access any data in the file prior to this address. When you create a mmap\_t object (but haven't yet opened a memory mapped file), or after you close the memory-mapped file (using the close method described below), the mmap\_t class initializes this field with NULL.

**fileSize:dword;**

This field holds the size of the file when it is mapped into memory. The memory mapping module initializes this field with zero when you don't have a currently opened memory-mapped file.

**endFilePtr:string;**

This field holds a pointer to the first byte beyond the end of the file in memory. You must not access any data in the file equal to or beyond this address. When you create a mmap\_t object (but haven't yet opened a memory mapped file), or after you close the memory-mapped file (using the close method described below), the mmap\_t class initializes this field with NULL.

The mmap\_t class also contains several private fields. Your applications must not modify the values of these private fields. The class does provide accessor methods if you wish to test the values of these private fields.

### 22.3 Class Procedures and Methods

Because the HLA stdlib implements the mmap\_t functions as a class, this document will not provide low-level calling sequence examples (which aren't especially practical for object-oriented function calls). Those who insist on making low-level calls to these functions should consult the HLA reference manual for information on making direct (low-level) calls to object-oriented functions.

```
procedure mmap_t.create(); @returns( "ESI" );
```

This procedure is the static class constructor. If you call this procedure using the class name (i.e., `mmap_t.create();`) then this constructor will allocate storage for a new `mmap_t` object on the heap, initialize that object, and return a pointer to the object in the ESI register. If you call this procedure via an object variable reference (e.g., `mmapVar.create();`) then this procedure will simply initialize the fields of that object.

As with all objects in HLA, you must call the `mmap_t.create` constructor before using the object. Failure to do so will cause the system to crash whenever you attempt to call any of this class' methods.

HLA high-level calling sequence example:

```
mmap_t.create( );
mov( esi, mmapObjPtr );
```

```
method mmap_t.destroy();
```

This is the class destructor. It deinitializes the `mmap_t` object, closes any memory-mapped file left open, and deallocates the storage for the object if it was allocated on the heap. Note that you should not rely upon the destructor to close your memory-mapped files - you should always explicitly call the `mmap_t.close` method to do this.

Because `mmap_t.destroy` is a method, you must only call this function after initializing some `mmap_t` object and you must only call this function via the object invocation mechanism. If you try to call `mmap_t.destroy` on an uninitialized `mmap_t` object, or if you try to call `mmap_t.destroy` directly, you will likely cause a program crash.

HLA high-level calling sequence examples:

```
mmapObjPtr.destroy( );
mmapStaticVar.destroy( );
```

```
method mmap_t.openNew( filename:string; maxSize:dword );
```

This method opens a new memory-mapped file. The `filename` parameter specifies the name of the file on the disk. If the file already exists, this call will delete the file before opening a new file by that name. The `filename` string must be a valid pathname. The `maxSize` parameter specifies the size of the file (in bytes) that this call will create. You must specify the size of the memory-mapped file when you open it. This method updates the object's fields, including the `filePtr`, `endFilePtr`, and `fileSize` fields. This procedure does not return the pointer to the file in EAX, use the `filePtr` field to obtain the address of the mapped file object. Note that this method always opens the file for reading and writing.

Because `mmap_t.openNew` is a method, you must only call this function after initializing some `mmap_t` object and you must only call this function via the object invocation mechanism. If you try to call `mmap_t.openNew` on an uninitialized `mmap_t` object, or if you try to call `mmap_t.openNew` directly, you will likely cause a program crash.

HLA high-level calling sequence examples:

```
mmapObjPtr.openNew( "AMemMappedFile", 8192 );
mmapStaticVar.openNew( "AnewFile", 16384 );
```

```
method mmap_t.open( filename:string; Access:dword );
```

This method maps an existing file into the process' address space. The `filename` parameter is a string specifying the pathname of the file to open. The `Access` parameter is either `fileio.r` or `fileio.rw` and specifies whether you're opening the file as a read-only or read/write object. This call maps the entire file into the process' address space (assuming the file is small enough to fit into the address space, of course). This method call initializes the `filePtr`, `endFilePtr`, and `fileSize` fields of the object as appropriate for the file.

Because `mmap_t.open` is a method, you must only call this function after initializing some `mmap_t` object and you must only call this function via the object invocation mechanism. If you try to call `mmap_t.open` on an uninitialized `mmap_t` object, or if you try to call `mmap_t.open` directly, you will likely cause a program crash.



HLA high-level calling sequence examples:

```
mmapObjPtr.open( "AMemMappedFile", fileio.r );
mmapStaticVar.open( "AnExistingFile", fileio.rw );
```

#### **method mmap\_t.close();**

This method unmaps the file and closes it. It also resets the object's fields to their default values (e.g., filePtr=NULL, endFilePtr=NULL, and fileSize=0). You may not access data in the memory mapped file after closing the file. Note that you may re-open the same (or a different) file using *mmap\_t.open* or *mmap\_t.openNew* after you close a file (and you don't need to call *mmap\_t.create* unless you also call *mmap\_t.destroy* after calling *mmap\_t.close*).

Because *mmap\_t.close* is a method, you must only call this function after initializing some *mmap\_t* object and you must only call this function via the object invocation mechanism. If you try to call *mmap\_t.close* on an uninitialized *mmap\_t* object, or if you try to call *mmap\_t.close* directly, you will likely cause a program crash.

HLA high-level calling sequence examples:

```
mmapObjPtr.close();
mmapStaticVar.close();
```

#### **method mmap\_t.getFileName();**

This is an accessor function that returns the filename string (pointer) in the EAX register. The program must not modify this string in any way. Note that this is a pointer to the string data held in the object itself, this is not a copy of the string. You should make a copy of this string if you intend to modify its data.

Because *mmap\_t.getFileName* is a method, you must only call this function after initializing some *mmap\_t* object and you must only call this function via the object invocation mechanism. If you try to call *mmap\_t.getFileName* on an uninitialized *mmap\_t* object, or if you try to call *mmap\_t.getFileName* directly, you will likely cause a program crash.

HLA high-level calling sequence example:

```
mmapObjPtr.getFileName();
mov( eax, fileNameString );
```

#### **method mmap\_t.getOpen();**

This is an accessor function that returns a boolean value in AL: false if the file is not currently open, true if there is a file mapped into the process' address space. This field is only valid after you call *mmap\_t.create*.

Because *mmap\_t.getOpen* is a method, you must only call this function after initializing some *mmap\_t* object and you must only call this function via the object invocation mechanism. If you try to call *mmap\_t.getOpen* on an uninitialized *mmap\_t* object, or if you try to call *mmap\_t.getOpen* directly, you will likely cause a program crash.

HLA high-level calling sequence example:

```
mmapObjPtr.getOpen();
if( al ) then

    // Do something if the file is open

endif;
```

```
method mmap_t.getMalloc();
```

This is an accessor function that returns true if the object is allocated dynamically on the heap, it returns false if the object is a static or automatic variable.

Because *mmap\_t.getMalloc* is a method, you must only call this function after initializing some *mmap\_t* object and you must only call this function via the object invocation mechanism. If you try to call *mmap\_t.getMalloc* on an uninitialized *mmap\_t* object, or if you try to call *mmap\_t.getMalloc* directly, you will likely cause a program crash.

HLA high-level calling sequence example:

```
mmapObjPtr.getOpen();  
if( al ) then  
  
    // Do something if the file is open  
  
endif;
```

## 23 Memory (memory.hhf)

The memory unit (header file is `memory.hhf`) contains the routines used to allocate and deallocate dynamic storage on the heap. There are a set of routines that allocate storage for general objects and a set of routines used to specifically allocate storage for strings.

As of HLA v1.69, the "allocation granularity" is eight bytes (that is, these routines always allocate data in multiples of eight byte chunks) and there is a 24-byte metadata overhead associated with each allocation. Therefore, you should avoid doing a large number of small allocations if you want to use memory efficiently. Note that these values are subject to change in future versions of the library.

These memory allocation routines associate a *reference counter* with each block. Whenever you first allocate a block on the heap, the reference counter is initialized with one. A *"mem.newref"* call instructs the heap management routines to increment this reference counter. The reference counter tracks how many different pointers in an application are referring to a single block of memory in the heap. When you call the *mem.free* routine to return storage to the heap, the heap management code will decrement the reference counter and only free up the storage when the reference counter decrements to zero. This can help avoid dangling pointers if you use the *mem.newref* routine in an appropriate fashion.

**A Note About Thread Safety:** The memory management routines maintain a couple of static global variables that track free and in-use blocks of memory. Currently, these values apply to all threads in a process. As such, the current implementation is not thread-safe. When the process module is added to the standard library, the memory management system will be modified to be thread safe. Until then, you should explicitly synchronize access to the HLA memory manager if you are writing multi-threaded applications.

### 23.1 Memory Module

To call functions in the Memory module, you must include one of the following statements in your HLA application:

```
#include( "memory.hhf" )
or
#include( "stdlib.hhf" )
```

Note: be sure to read the chapter on "Passing Parameters to Standard Library Routines" (*parmpassing.rtf*) before reading this chapter.

### 23.2 Deprecated Names

The HLA Standard Library has inherited some older, deprecated, names from the HLA *stdlib* v1.x. If you look at the include files for the Standard Library, those names might still be present. This document, however, will not describe those deprecated names from the v1.x library.

### 23.3 Generic Memory Allocation

The following functions allocate, deallocate, and operate on blocks of memory that may contain arbitrary data.

**mem.alloc overloads mem.alloc1 and mem.alloc2**

If you invoke *mem.alloc* with one parameter, it calls *mem.alloc1*; if you call *mem.alloc* with two parameters, it calls *mem.alloc2*.

```
procedure mem.alloc1( size:dword ); @returns( "eax" );
```

The *mem.alloc1* routine allocates the requested number of bytes. If successful, this routine returns a pointer to the allocated storage in the EAX register. This routine raises an *ex.MemoryAllocationFailure* exception or an *ex.MemoryAllocationCorruption* exception if it fails. Note that this function does not initialize the block of memory to any particular value when it allocates it. In particular, do not count on this function setting the block of memory to zeros.

HLA high-level calling sequence example:

```
mem.alloc1( 1024 );
mov( eax, memBlkPtr );
```

HLA low-level calling sequence example:

```
pushd( 1024 );
call mem.alloc1;
mov( eax, memBlkPtr );
```

```
procedure mem.alloc2( size:dword; callback:thunk );
@returns( "eax" );
```

This function is very similar to *mem.alloc1* with one major difference: after allocating the block, it will call the *callback* thunk. This allows the caller to track memory usage, initialize the memory block, or perform any other activity before returning from *mem.alloc2*. On entry into the thunk, ECX will contain the block size and EAX will point at the memory block. The direction flag will be clear. Anything you do in the thunk is entirely up to you, but you will want to return a pointer to an appropriately sized memory block in the EAX register. You can use the other registers (ebx, ecx, edx, esi, and edi) as you see fit.

HLA high-level calling sequence example:

```
mem.alloc2( 2048, thunk #{ call savePtrInEAX; }# );
mov( eax, memBlkPtr );
```

HLA low-level calling sequence example:

```
pushd( 2048 );
push( ebp );// Thunk pointer
pushd( &thunkCode );
jmp callrealloc2;
thunkCode:
    call savePtrInEAX;
    ret();

callrealloc2:
call mem.alloc2;
mov( eax, memBlkPtr );
```

```
procedure mem.zalloc( size:dword ); @returns( "eax" );
```

The *mem.zalloc* routine allocates the requested number of bytes and zeros out the data storage allocated. If successful, this routine returns a pointer to the allocated storage in the EAX register. This routine raises an *ex.MemoryAllocationFailure* exception or an *ex.MemoryAllocationCorruption* exception if it fails.

HLA high-level calling sequence example:

```
mem.zalloc( 1024 );
mov( eax, memBlkPtr );
```

HLA low-level calling sequence example:

```
pushd( 1024 );
call mem.zalloc;
mov( eax, memBlkPtr );
```

```
procedure mem.free( memptr:dword );
```

This function frees up storage previously allocated by the *mem.alloc* routine. A pointer returned from *mem.alloc* must be passed as the parameter to this function. This routine actually decrements a *reference counter* and only frees the storage when the reference counter becomes zero. See the discussion of *mem.newref* for more details.

HLA high-level calling sequence example:

```
mem.free( memBlkPtr );
```

HLA low-level calling sequence example:

```
push( memBlkPtr );
call mem.free;
```

**mem.realloc overloads mem.realloc1 and mem.realloc2**

If you invoke *mem.realloc* with two parameters, it calls *mem.realloc1*; if you call *mem.realloc* with three parameters, it calls *mem.realloc2*.

```
procedure mem.realloc1( memptr:dword; newsize:dword ); @returns( "eax" );
```

The *mem.realloc1* routine resizes a previous allocated block of memory. The first parameter is the pointer to the original block, the second parameter is the new size. If the new block is smaller, this routine truncates the data beyond the new size. If the new block is larger, this routine will copy the data if it cannot expand the block in-place.

If the address of the block does not change, then the block created by *mem.realloc1* inherits the reference counter value from the original block. However, if the *mem.realloc1* function must create a new block and copy the data to that new block, then the reference counter of the new block is set to one. If the reference counter of the original block was not one prior to the *realloc* operation, then the system simply decrements the original reference counter and does not deallocate the original storage. It is important to realize that the *mem.realloc1* operation may leave two allocated blocks and any previous pointers (noted by *mem.newref* calls) are still valid and still point at the original data. The pointer returned by *mem.realloc1* points at the new block.

HLA high-level calling sequence example:

```
mem.realloc1( memBlkPtr, 2048 );
mov( eax, memBlkPtr );
```

HLA low-level calling sequence example:

```
push( memBlkPtr );
pushd( 2048 );
call mem.realloc1;
mov( eax, memBlkPtr );
```

```
procedure mem.realloc2( memptr:dword; newsize:dword; copycallback:thunk );  
@returns( "eax" );
```

This function is very similar to *mem.realloc1* with one major difference: if, during the reallocation operation, *mem.realloc2* needs to copy a block of data because it cannot expand the existing block in-place, it will call the *copycallback* thunk to handle the copy operation. This allows the caller to readjust application-dependent pointers and do other activities if the block has to be moved during a *realloc* operation. On entry into the thunk, ECX will contain the block size, ESI will point at the source block, and EDI will point at the destination block. The direction flag will be clear and you can assume that the blocks do not overlap. You should, at the very least, execute a "rep.movsb;" instruction to copy the source block to the destination block.

Anything else you do in the thunk is entirely up to you, but typically, you will want to adjust any pointers in your application that point at the source block so that they point at the destination block.

HLA high-level calling sequence example:

```
mem.realloc2( memBlkPtr, 2048, thunk #{ rep.movsb }# );
mov( eax, memBlkPtr );
```

HLA low-level calling sequence example:

```
push( memBlkPtr );
pushd( 2048 );
push( ebp );// Thunk pointer
pushd( &thunkCode );
jmp callrealloc2;
thunkCode:
    rep.movsb();
    ret();

callrealloc2:
call mem.realloc2;
mov( eax, memBlkPtr );
```

**#macro mem.talloc( size );      (returns "eax" as macro result)**

This is a macro that "temporarily" allocates the specified storage. This macro allocates the specified storage on the stack and returns the address of the storage (i.e., the ESP value) in the EAX register. The address is always dword aligned; *mem.talloc* will allocate up to three additional bytes to ensure dword alignment.

You may use the *mem.talloc* call anywhere a single instruction is legal (including using *mem.talloc* as an operand to another instruction).

There is no corresponding "tfree" routine since leaving the current procedure automatically deallocates the storage. That is, when a standard procedure exits, it resets the stack pointer, automatically removing the *mem.talloc*'d data. If you would like to explicitly free the data, then you should save the value of ESP prior to calling *mem.talloc* and this restore ESP from this saved value when you want to "free" the storage.

**Warning:** in order for a function to properly free the storage allocated by *mem.talloc*, the function must have a standard activation record or must otherwise restore ESP to the value it held prior to the invocation of *mem.talloc*. HLA procedures that generate a standard activation record (e.g., those that don't have the @noframe option) do this automatically. But if you write a procedure that has the @noframe option, you must take responsibility for restoring ESP's value to deallocate the storage set aside by *mem.talloc*.

Obviously, you cannot continue referencing the data allocated by *mem.talloc* once the enclosing procedure returns.

HLA high-level calling sequence example:

```
mem.talloc( 128 );
mov( eax, memBlkPtr );
```

Note: Because this is a macro, there is no low-level calling sequence.

**procedure mem.isInHeap( memptr:dword );**

This function returns false (NULL) in EAX if the *memptr* parameter does not point at a valid (allocated) object on the heap. It returns a pointer to the start of the data block on the heap if *memptr* does point within the data area of a valid block. You can use this function to determine whether an object was previously allocated via a call to *mem.alloc* (and should be free'd via a call to *mem.free*). Note that this function only returns non-NULL if the block is currently allocated. If you've free'd all instances of the block, this function will return NULL. In older versions of this routine, the function simply returned true or false. Assuming older code treated false as zero and true as anything else, that code will continue to function with this new version of the routine.

HLA high-level calling sequence example:

```
mem.isInHeap( memBlkPtr );
if( eax <> NULL ) then

    mem.free( eax );

endif;
```

HLA low-level calling sequence example:

```
push( memBlkPtr );
call mem.isInHeap;
test( eax, eax );
jz noFree;

    push( eax );
    call mem.free;

noFree:
```

#### **procedure mem.size( memptr:dword );**

This function returns the amount of storage allocated in the block pointed at by *memptr*. The value of *memptr* must be a value returned by *mem.alloc*, *mem.realloc*, or *mem.realloc2*. Note that the value that *mem.size* returns might be slightly larger than the original request. This function returns the actual size of the allocated block, including any padding bytes added to the end of the block for alignment purposes.

HLA high-level calling sequence example:

```
mem.size( memBlkPtr );
mov( eax, blockSize );
```

HLA low-level calling sequence example:

```
push( memBlkPtr );
call mem.size;
mov( eax, blockSize );
```

#### **procedure mem.stat;**

This function returns statistics concerning the heap space in use by the memory allocation routines. This function returns the following values:

- EAX - Total amount of space currently in use by the heap (this may not be contiguous!).
- EBX - Total amount of free space in the heap.
- ECX - Largest block of contiguous free space in the heap.
- EDX - Number of blocks on the heap (free and in use).
- EDI - Number of free blocks on the heap.

Note that the value in EBX, the total amount of free space in the heap, does not indicate the maximum amount of space that you can allocate. This simply indicates the amount of space that was previously allocated and has been freed. Generally, it is quite possible to allocate more storage than is available in the heap at any one time. Indeed, prior to the first *mem.alloc* operation, the *mem.stat* function will return zero in all these registers.

HLA high-level calling sequence example:

```
mem.stat();
mov( eax, spaceInUse );
mov( ebx, freeSpace );
mov( ecx, largestBlock );
mov( edx, numBlocks );
mov( edi, numFreeBlocks );
```

HLA low-level calling sequence example:

```
call mem.stat;
mov( eax, spaceInUse );
mov( ebx, freeSpace );
mov( ecx, largestBlock );
mov( edx, numBlocks );
mov( edi, numFreeBlocks );
```

### **mem.newref( memblk:dword );**

This funtion increments a *reference counter* for the memory block whose address you pass as the parameter (this must be a block allocated by *mem.alloc*). The heap routines will not deallocate storage for a block of memory until you've called *mem.free* the number of times specified by the reference counter. The *mem.alloc* call initializes the reference counter to one, calls to *mem.newref* increment this value by one, calls to *mem.free* decrement this value by one (and frees the storage once the reference counter hits zero).

HLA high-level calling sequence example:

```
mem.newref( memPtr );
```

HLA low-level calling sequence example:

```
push( memPtr );
call mem.newref;
```

### **mem.getref( memblk:dword );**

This funtion returns the *reference counter* value for the specified memory block. This function raises an *ex.PointerNotInHeap* exception if *memblk* does not point within a valid memory block. Note that if the block has been deallocated, this function returns zero, it does not raise an exception.

HLA high-level calling sequence example:

```
mem.getref( memPtr );
mov( eax, refCnt );
```

HLA low-level calling sequence example:

```
push( memPtr );
call mem.getref;
mov( eax, refCnt );
```



**iterator mem.blockInHeap;**

This is an iterator (used in a foreach loop) that returns the following information for each block (free and in-use) in the heap, one block per iteration:

EAX - Size of block

EBX - Address of data block

ECX - Reference count for block

This function is mainly intended for debugging purposes.

HLA high-level calling sequence example:

```
foreach mem.blockInHeap() do

    stdout.put
    (
        "size=$", eax,
        " adrs=$", ebx,
        " refcnt=$", ecx,
        nl
    );

endfor;
```

HLA low-level calling sequence example:

```
pushd( &endLoopBody );
call mem.blockInHeap;
stdout.put
(
    "size=$", eax,
    " adrs=$", ebx,
    " refcnt=$", ecx,
    nl
);
ret();

endLoopBody:
```

**iterator mem.allocBlockInHeap;**

This iterator is similar to mem.blockInHeap except it only iterates over the allocated blocks in the heap.

This function is mainly intended for debugging purposes.

HLA high-level calling sequence example:

```
foreach mem.allocBlockInHeap() do

    stdout.put
    (
        "size=$", eax,
        " adrs=$", ebx,
        " refcnt=$", ecx,
        nl
    );

endfor;
```

HLA low-level calling sequence example:

```

pushd( &endLoopBody );
call mem.allocBlockInHeap;
stdout.put
(
    "size=$", eax,
    " adrs=$", ebx,
    " refcnt=$", ecx,
    nl
);
ret();

endLoopBody:

```

#### **iterator mem.freeBlockInHeap;**

This iterator is similar to mem.blockInHeap except it only iterates over the free blocks in the heap. This function is mainly intended for debugging purposes.

HLA high-level calling sequence example:

```

foreach mem.freeBlockInHeap() do

    stdout.put
    (
        "size=$", eax,
        " adrs=$", ebx,
        " refcnt=$", ecx,
        nl
    );

endfor;

```

HLA low-level calling sequence example:

```

pushd( &endLoopBody );
call mem.freeBlockInHeap;
stdout.put
(
    "size=$", eax,
    " adrs=$", ebx,
    " refcnt=$", ecx,
    nl
);
ret();

endLoopBody:

```

## **23.4 String Memory Allocation**

The memory-related functions in this category are used to allocate, deallocate, and manipulate dynamic string data. The main difference between these functions and the "standard" memory allocation functions is the pointer values these function manipulate. Because HLA string pointers must contain an address that is eight bytes into the string data structure (unlike standard memory allocation functions that work with pointers that point at the beginning of the memory block), these string functions automatically add or subtract that offset.

Because the calls to these functions are identical to the standard memory functions boasting the same names, please see the calling sequence examples given earlier.

```
procedure str.alloc( size:dword ); @returns( "eax" );
procedure str.realloc( strPtr:dword; size:dword ); @returns( "eax" );
procedure str.free( strPtr:dword );
procedure str.isInHeap( strPtr:dword ); @returns( "eax" );
```

The string allocation routines are used just like the general memory allocation routines except they allocate storage for a string variable and initialize the string object's *maxLength* and *length* fields. They return a pointer to the first character position of the string's data (that is, the address of the byte just beyond the *maxLength* and *length* fields). Note that the *str.isInHeap* function returns a pointer to the start of the string's data (the first character in the string) if it determines that the string has been allocated on the heap. See the discussion of *mem.realloc* to understand how *str.realloc* affects the reference counter for a string on the heap.

```
#macro str.talloc( size );      (returns pointer to new string in EAX ).
```

This is a macro that initializes storage on the stack for a string capable of holding *size* characters. This routine has the same benefits and drawbacks as the *mem.talloc* routine.

Note that the *size* parameter is the actual number of characters needed. the *str.talloc* routine automatically bumps this value up by nine to make room for the *length*, *maxLength*, and zero terminator fields of the string object. This macro also ensures that the stack (and, therefore, the string) is dword aligned in memory (it does this by adding up to three additional bytes to the string).

```
procedure str.newref( strPtr:dword );
```

This function increments a *reference counter* for the memory block whose address you pass as the parameter (this must be a block allocated by *str.alloc*). The heap routines will not deallocate storage for a block of memory until you've called *str.free* the number of times specified by the reference counter. The *str.alloc* call initializes the reference counter to one, calls to *str.newref* increment this value by one, calls to *str.free* decrement this value by one (and frees the storage once the reference counter hits zero).

```
str.getref( strPtr:dword );
```

This function returns the *reference counter* value for the specified string memory block. This function raises an exception if *strPtr* does not point within a valid memory block allocated for a string. Note that if the string has been deallocated, this function returns zero, it does not raise an exception.



## 24 OS Module (os.hhf)

The OS module contains a couple functions that do OS-related tasks.

### 24.1 The OS Module

To use the OS functions in your application, you will need to include one of the following statements at the beginning of your HLA application:

```
#include( "os.hhf" )
or
#include( "stdlib.hhf" )
```

### 24.2 Executing Shell Commands

```
procedure os.system( cmdStr:string );
```

The *os.system* function executes a single program and waits for the execution of that command before returning. Here is the syntax for the *os.system* call:

```
os.system( "system command" );
```

The string you pass as the single parameter roughly corresponds to a command shell command (e.g., the Windows command line prompt or the Linux/FreeBSD/MacOS Shell prompt). This consists of the program name followed by any command line parameters, separated by spaces.

The first thing to note about this function is that the results are system-specific. Although this function is available in all operating systems that the HLA Standard Library supports, the semantics of the commands you pass to this function vary by operating system. Therefore, programs that call this function will not usually be portable between operating systems.

Special notes for Windows users: the *os.system* function does not directly allow the execution of intrinsic (built-in) cmd.exe commands. If you want to execute a command like DIR, CD, MD, etc., that aren't actual programs, but simply commands that cmd.exe executes directly, you have to run an instance of the command interpreter to pull this off, e.g.,

```
os.system( "cmd /C dir" ); // Executes 'DOS' directory command
```

Please see the description of the Windows "cmd.exe" program for more details (type "help cmd" at the command line prompt). Also note that Windows will use the current PATH environment variable to locate the executable program, if it is not in the current subdirectory.

Special notes for Linux/FreeBSD users: If the program name appearing at the beginning of the string does not specify the path to a file that Linux/FreeBSD can find, Linux/FreeBSD will prefix the name with "/bin/" and then "/usr/bin/" in an attempt to locate the file.

The function fails silently if it cannot find or execute the specified program.

HLA high-level calling sequence examples:

```
os.system( "ls" ); // Under Linux or FreeBSD
os.system( "HLA t.hla" );// Runs HLA compiler on the "t.hla" file.
```

HLA low-level calling sequence examples:

```
static
    cmd:string := "HLA t.hla";
    .
    .
    .
    push( cmd );
    call os.system;
```

## 24.3 Delaying Program Execution

The OS module provides two functions that will suspend (put to sleep) a process for a short period of time. The first function (`sleep`) lets you specify the suspension time in seconds, the other (`mSleep`) lets you specify the time in milliseconds. It is important for you to realize that the underlying operating systems do not guarantee that the delay will be exactly equivalent to the duration you specify. Most operating systems only guarantee that they will suspend the program for *at least* as long as you specify – they might actually delay the program even longer.

```
procedure os.sleep( secs:dword );
```

This function suspends the program for at least *secs* seconds. After at least *secs* seconds have transpired, the OS will place the program back into the run queue and the process will begin execution after the call to *os.sleep* on the next regularly-scheduled time quantum.

Specifying an argument value of zero may have no effect (that is, the *os.sleep* call may immediately return), but many operating systems will cause the current process to give up the remainder of its time slice when you call *os.sleep* in this fashion. You should, however, not count on such semantics in your program.

```
procedure os.mSleep( msecs:dword );
```

This function suspends the program for at least *msecs* milliseconds. After at least *msecs* milliseconds have transpired, the OS will place the program back into the run queue and the process will begin execution after the call to *os.sleep* on the next regularly-scheduled time quantum.

Specifying an argument value of zero may have no effect (that is, the *os.sleep* call may immediately return), but many operating systems will cause the current process to give up the remainder of its time slice when you call *os.sleep* in this fashion. You should, however, not count on such semantics in your program.

## 25 Patterns Module (patterns.hhf)

The HLA Standard Library provides a set of string/pattern matching routines that are similar in use to those provided by the SNOBOL4 and Icon programming languages. These pattern matching routines support recursion and backtracking, allowing the specification of context-free grammars as well as regular expressions.

**Note:** Because many of the "functions" in the pattern-matching library are actually macro invocations, this document does not provide examples of low-level pattern-matching function calls.

**Warning:** unlike most HLA Standard Library functions, the pattern matching functions do not preserve all the registers they modify. In fact, EDX is the only register whose value may be preserved; almost all the other registers are used by the pattern matching code and you should expect their values to be modified by the pattern matching functions whenever you call them.

### 25.1 The Patterns Module

To use the pattern functions in your application, you will need to include one of the following statements at the beginning of your HLA application:

```
#include( "patterns.hhf" )
or
#include( "stdlib.hhf" )
```

### 25.2 An Introduction to Pattern Matching (a tutorial)

The HLA pattern matching library scans for patterns of characters within a string or within some sequence of characters. A pattern matching operation consists of a sequence of pattern matching commands that execute on the sequence. The result of a pattern matching operation is either *success* (meaning all the pattern matching commands succeeded) or *failure* (meaning at least one of the commands failed to match). The success or failure of a pattern matching operation directs program execution to one of two different locations in the code (not unlike an IF/ELSE/ENDIF statement) so the program can perform different operations based on the success or failure of a pattern match.

You must understand that the HLA Standard Library pattern matching functions don't return true or false that you can test in a conditional expression (e.g., in an IF or WHILE statement). Instead, the pattern matching functions and macros actually introduce a *new control structure* in the HLA language. Within this control structure, the successful execution of each pattern matching operation allows the program to continue execution with the next successive statement in the control structure. However, if the pattern matching operation fails, then control transfers to a different location in the pattern matching control construct. In a sense, this is very similar to HLA's *try..exception..endtry* statement. A failed pattern matching operation transfers control to some distinct failure location (just like an exception occurring in a *try..endtry* block); successful pattern matching operations fall through to the next command or, if all commands in a pattern matching command sequence are successful, control transfers to the first statement after the pattern matching control structure.

The pattern matching control sequence is delimited by the *pat.match* and *pat.endmatch* macro invocations. Between these two statements, exactly one *pat.if\_failure* macro invocation must appear. The template of a pattern matching statement is the following:

```
pat.match( <<character sequence to match>> );

    << Sequence of match operations>>

    << Code to execute on a successful match >>

pat.if_failure

    << Code to execute if the match fails >>

pat.endmatch;
```

The "sequence of match operations" appearing in this control structure is a set of zero or more pattern matching function calls. As noted above, if a given function succeeds, the control falls through to the next command in the control structure (or through to the "code to execute on a successful match" if all of the matching commands succeed). If a match operation fails, the the program immediately transfers control to the code following the *pat.if\_failure* statement.

The *pat.match* statement supports two difference syntaxes. The first form accepts a single HLA string object as a parameter. This form is invoked thusly:

```
pat.match( StringValue );
```

Technically, you could supply a string variable or a string constant as this argument. However, it would never make any sense (other than for testing or demonstration purposes) to supply a literal string constant as this argument. The purpose of the pattern matching functions is to determine if some unknown string matches a given pattern. If the string's value is known while you're writing the program, there really isn't any need to do the pattern matching operation – you can do the pattern matching operation in your head and skip the execution of the code. Nevertheless, many of the examples in this document will use literal string constants as the test string in order to make the examples easier to understand.

The second form of the *pat.match* statement expects two arguments. The first is a pointer to the first character of some character sequence you wish to match and the second argument is a pointer to the first byte beyond the end of the character sequence you wish to match. This invocation takes the following form:

```
pat.match( StartOfSequence, EndOfSequence );
```

Note that internally, the *pat.match* macro actually uses the start and end of sequence pointers. If you pass the *pat.match* function a single string argument, *pat.match* uses the string pointer as the *StartOfSequence* pointer and it adds the strings length to the *StartOfSequence* value to obtain the *EndOfSequence* address. For the sake of discussion, we'll call the string (or sequence of characters) we're trying to match the *match sequence*.

During a pattern matching operation, there are three important pointers the functions use: a pointer to the first character of the character sequence, a pointer to the first byte beyond the character sequence, and a *cursor* pointer that points at the next character under consideration. When you invoke *pat.match*, the macro begins by initializing the cursor with the address of the first character in the match sequence (e.g., the *StartOfSequence* value). The pattern matching commands operate on the character data at the current cursor position through the end of the sequence (that is, up to the byte before the address held in the end of sequence pointer). If the cursor's value is ever greater than or equal to the end of sequence value and the program attempts to execute a pattern matching function that would advance the cursor, then the pattern matching operation fails (and control transfers to the *pat.if\_failure* statement).

Let's consider a concrete example. The *pat.oneChar* function accepts a single character argument. If the cursor's value is less than the end of sequence value and the character that the cursor points at matches *pat.oneChar*'s argument, then the *pat.oneChar* function succeeds and increments the cursor value to skip over the character in the sequence that it matched. The following pattern matching operation succeeds and prints "Encountered 'c'":

```
pat.match( "c" );

pat.oneChar( 'c' );
stdout.put( "Encountered 'c'" nl );

pat.if_failure

stdout.put( "Failed to match 'c'" nl );

pat.endmatch;
```

The following code, however, prints "Failed to match 'c'" because the cursor (initialized with the address of the 'd' character) doesn't point at a byte containing 'c':

```
pat.match( "d" );

pat.oneChar( 'c' );
stdout.put( "Encountered 'c'" nl );

pat.if_failure

stdout.put( "Failed to match 'c'" nl );

pat.endmatch;
```



Whenever a pattern matching function such as *pat.oneChar* succeeds, it advances the cursor over the character(s) it matches. Upon return from the pattern matching function, any successive calls to a pattern matching function will attempt to match the character(s) immediately after those already matched. For example, consider the following pattern matching construct:

```
pat.match( "cd" );

pat.oneChar( 'c' );
pat.oneChar( 'd' );
stdout.put( "Encountered 'cd'" nl );

pat.if_failure

stdout.put( "Failed to match 'cd'" nl );

pat.endmatch;
```

The first call to *pat.oneChar* matches the 'c' in the match sequence and advances the cursor by one position (so that it now points at the 'd' character). The second call to *pat.oneChar* in this example matches the 'd' character in the match sequence, increments the cursor to point at the byte beyond the "cd" string, and then returns and prints "Encountered 'cd'".

As noted earlier, if a sequence of pattern matching commands advances the cursor to the point it "runs off the end" of the character sequence, then the pattern matching sequence fails. The following example demonstrates this (it will print "Failed to match 'cd'"):

```
pat.match( "c" );

pat.oneChar( 'c' );
pat.oneChar( 'd' );
stdout.put( "Encountered 'cd'" nl );

pat.if_failure

stdout.put( "Failed to match 'cd'" nl );

pat.endmatch;
```

Note, however, that a pattern matching operation does not fail if it doesn't consume all the characters in the match sequence (that is, it doesn't advance the cursor to the end of the match sequence). The following example succeeds and prints "Encountered 'c'" even though it doesn't consume all the characters in the match sequence:

```
pat.match( "cd" );

pat.oneChar( 'c' );
stdout.put( "Encountered 'c'" nl );

pat.if_failure

stdout.put( "Failed to match 'c'" nl );

pat.endmatch;
```

If you're wondering why this shouldn't fail, just note that you can build up complex pattern matching function by making nested and recursive *pat.match* invocations, in such cases you don't want to fail if you've not reached the end of the match sequence because further calls to *pat.match* may handle the remaining characters in the match sequence. In those cases where you really do want to fail if you don't match the entire match sequence, the HLA pattern matching module provides a special function, *pat.EOS*, that explicitly checks for the

end of the match sequence. The following modification to the previous example will display "Failed to match 'c'":

```
pat.match( "cd" );

pat.oneChar( 'c' );
pat.EOS();
stdout.put( "Encountered 'c'" nl );

pat.if_failure

stdout.put( "Failed to match 'c'" nl );

pat.endmatch;
```

Earlier, this document suggested that a *pat.match..pat.if\_failure..pat.endmatch* statement was similar to an IF/ELSE/ENDIF statement insofar as there are two sections of code where you can wind up based on success or failure of the match. In fact, the *pat.match..pat.endmatch* statement is actually closer to an IF/ELSEIF/ELSE/ENDIF statement. If the sequence of pattern matching operations immediately after the *pat.match* statement fail, it is possible to transfer control to another pattern matching operation that will try to succeed. This is known as *alternation* (that is, seeking an alternative match). If the *pat.alternate* statement appears between the *pat.match* and the *pat.if\_failure*, then this will supply an alternate pattern matching sequence to try if the main matching sequence fails. Only if both the main and alternate patterns fail will the entire pattern matching operation fail. Consider the following example:

```
pat.match( "cd" );

pat.oneChar( 'c' );
pat.EOS();
stdout.put( "Encountered 'c'" nl );

pat.alternate

pat.oneChar( 'c' );
pat.oneChar( 'd' );
pat.EOS();
stdout.put( "Encountered 'cd'" nl );

pat.if_failure

stdout.put( "Failed to match 'c' or 'cd'" nl );

pat.endmatch;
```

This pattern succeeds and prints "Encountered 'cd'". It begins by trying to match against 'c' (which succeeds) followed by the end of string (which fails). When failure occurs, the *pat.match* statement resets the cursor to the start of the sequence (that is, to the beginning of the "cd" string) and transfers control to the *pat.alternate* statement). This sequence of match operations will match the 'c', the 'd', and the end of the string, and then print "Encountered 'cd'".

A *pat.match..pat.endmatch* statement can have any number of *pat.alternate* clauses in it (just as an IF/ELSEIF/ELSE statement can have any number of ELSEIF clauses). The *pat.match* statement will transfer control to the first *pat.alternate* section if the main pattern matching command set fails; it will transfer control to the second *pat.alternate* section if both the main pattern matching sequence and the first alternate sequence fail; a fourth *pat.alternate* section will execute if the main and first two alternate sections fail; etc. The *pat.if\_failure* section will only execute if the main section and all the alternate sections fail to match their patterns.

Note that the *pat.if\_failure* section must follow all the *pat.alternate* sections in the *pat.match..pat.endmatch* statement. HLA will report an error if any *pat.alternate* sections follow the *pat.if\_failure*. Also remember: the *pat.if\_failure* section is not optional. HLA will report an error if a *pat.if\_failure* section is not present in a *pat.match..pat.endmatch* statement.

## 25.3 Pattern Matching Functions Versus User Code

The discussion in the previous section may have led you to believe that a pattern matching section (either the main section or an alternate section) consisted of two parts: the pattern matching code sequence and the user code to execute upon successfully matching the pattern:

```
pat.match( <<character sequence to match>> );

    <<Sequence of match operations>>

    <<Code to execute on a successful match>>

pat.if_failure

    << Code to execute if the match fails >>

pat.endmatch;
```

In fact, there is no distinction between <<Sequence of match operations>> and <<Code to execute on a successful match>>. The program is going to execute the statements in a matching section until either failure occurs (in which case control transfers to the next *pat.alternate* section, or to the *pat.if\_failure* section if there is no alternate), or the execution sequence reaches a *pat.alternate* or *pat.if\_failure* statement (at which point control transfers to the first program statement following the *pat.endmatch* clause). The pattern matching functions themselves are really nothing more than 80x86 code that know how to transfer control to some failure clause if the matching function fails. So although most pattern matching statements are organized as described earlier (with the pattern matching operations appearing first and the statements to execute on a successful match occurring afterward), it is possible to inject standard machine instructions and other HLA statements between the pattern matching operations. *However, you must exercise extreme caution when doing so.*

Consider the following example;

```
pat.match( testString );

    pat.oneChar( 'c' );
    stdout.put( "Encountered 'c'" nl );
    pat.EOS();
    stdout.put( "Encountered EOS" nl );

    pat.alternate

    pat.oneChar( 'c' );
    stdout.put( "Encountered 'c'" nl );
    pat.oneChar( 'd' );
    stdout.put( "Encountered 'd'" nl );
    pat.EOS();
    stdout.put( "Encountered EOS" nl );

    pat.if_failure

    stdout.put( "Failed to match 'c' or 'cd' followed by EOS" nl );

pat.endmatch;
```

If testString turns out to have the value "c", then the main matching section succeeds and prints

```
Encountered 'c'
Encountered EOS
```

So far, so good. Now, however, suppose that testString holds the value "cd". In this case, the alternate section succeeds and the program prints the following:

```

Encountered 'c'
Encountered 'c'
Encountered 'd'
Encountered EOS

```

No, this is not a typographical error. Yes, it prints "Encountered 'c'" twice. This happens because the main pattern matching section doesn't fail until after it executes the *stdout.put* statement that prints "Encountered 'c'". Generally, failed matches should be *transparent*; that is, they should not affect the system by printing or changing values. This is why most pattern matching sequences appear before any user code (technically called the "semantic action") in a pattern matching sequence. You never want to do something that cannot be undone (such as print data to the console) should the pattern matching operation fail.

## 25.4 Register and Stack Usage in Pattern Matching Statements

During a pattern matching operation (that is, between the *pat.match* and *pat.endmatch* statements), the pattern matching code makes use of most of the 80x86's registers to maintain value such as the cursor, end of sequence pointer, and other values. Therefore, you cannot assume that any register values will be preserved across pattern matching function calls and, even more importantly, you must not play around with the register values between pattern matching function calls as these functions communicate between one another using the registers. Even the stack pointer is not sacrosanct. Many pattern matching functions will actually leave data on the stack upon return (to implement a facility known as *backtracking*, which you'll read about a little later). Therefore, you must exercise caution when mixing user statements and pattern matching statements in the same code sequence (that is, this is yet another good reason to put all your "semantic actions" *after* all the pattern matching operations). This section will discuss how the pattern matching code uses registers and and stack, so you can deal with the issue accordingly.

The *pat.match* statement initializes the ESI register with the cursor value (that is, the address of the first character in the match sequence) and EDI with the address of the byte just beyond the end of the match sequence (the *EndOfSequence* value). Whenever a pattern matching function successfully returns, EBX will contain the original cursor value (upon entry into that function) and ESI will contain the new cursor value (that is, it will point beyond all the characters that the function matched). Therefore, EBX..(ESI-1) will be the sequence of characters matched by the function.

Almost all pattern matching functions scramble the value in the EAX register prior to returning (actually, "scramble" is a bad term, most functions actually load the return address for the function into EAX and return by jumping indirectly through EAX's value rather than by executing a RET instruction). Many pattern matching functions modify ECX's value (e.g., for use as a "repeat count" when used with the string instructions). Most of the original Standard Library functions preserve the value held in the EDX register, but because the pattern matching library is extensible, it's dangerous to assume that EDX is preserved.

The *one* register you can count on being preserved is EBP. Upon return from a pattern matching function, you can count on EBP containing the address of your stack frame (assuming you use EBP for this purpose).

As noted earlier, many (most) pattern matching functions do not preserve the value of ESP when they return. In particular, most pattern matching functions actually leave data sitting on the stack when they return. This data may get used by later pattern matching functions should failure occur. Of course, the pattern matching code will eventually clean up after itself; in particular, when you execute the *pat.endmatch* statement the code will clean up the stack and leave it in the same state it was when the program executed the corresponding *pat.match* statement. There are two important implications of this:

You cannot use PUSH and POP instructions to preserve values across a pattern matching function. The following will *not* work:

```

pat.match( "c" );

push( eax );
pat.oneChar( 'c' );
pat.EOS();
pop( eax );

pat.if_failure

    stdout.put( "did not match 'c' " nl );

pat.endmatch;

```

The problem is that *pat.oneChar* and *pat.EOS* may leave extra data sitting on the stack when they return. Therefore, the `POP( EAX );` instruction will not be popping the EAX data originally pushed, instead it will pop off some data left on the stack by the pattern matching functions.

Of course, in a sequence of statements you write, that do not call any pattern matching functions (or anything else that doesn't preserve ESP's value), you may certainly use `PUSH` and `POP` in a traditional manner. In particular, if you put all your user code after all the pattern matching function calls, then you can use `PUSH` and `POP` to your heart's content. However, be aware that pushing and popping data round pattern matching function calls may not work as you expect.

Because the *pat.endmatch* clause is responsible for cleaning up the stack, removing any data left on the stack by pattern matching functions, you should never exit out of a *pat.match..pat.endmatch* statement by jumping out of the middle of the code to some label outside the *pat.match..pat.endmatch* sequence. For example, don't do the following:

```
pat.match( "c" );

push( eax );
pat.oneChar( 'c' );
jmp cIsGoodEnough;

pat.if_failure

    stdout.put( "did not match 'c'" nl );

pat.endmatch;

cIsGoodEnough:// Junk may be left on the stack here.
```

The one exception to this rule is exception-handling code. If an exception occurs in the *pat.match..pat.endmatch* statement, the exception handling system will automatically clean up the stack for you before transferring control to your exception handling code sequence. Other than exit by exception, the only way you should leave a *pat.match..pat.endmatch* statement is by "running off the end" of a pattern matching section (that is, by encountering a *pat.alternate* or *pat.if\_failure* clause during the normal sequential execution of the pattern matching section).

One other big piece of advice: avoid using any form of control structures, especially loop control structures, within the pattern matching sequence. In practice, there isn't much need to put a series of pattern matching functions inside a `WHILE` or `FOR` loop or inside an `IF` statement. As you'll discover, the HLA pattern matching module provides a rich variety of functions that automatically process repetitive data or conditionally match one sequence or another (e.g., by using alternation).

Ultimately, the best advice you can follow is to adhere to the original syntax given for the *pat.match..pat.endmatch* statement:

```
pat.match( <<character sequence to match>> );

    << Sequence of match operations>>

    << Code to execute on a successful match >>

pat.if_failure

    << Code to execute if the match fails >>

pat.endmatch;
```

That is, put all your pattern matching function calls at the beginning of a match section and put the "Code to execute on a successful match" (the "semantic action") after those function calls. Within the semantic action, you can feel free to write any 80x86 code you like (as long as it doesn't make any pattern matching function calls that are part of the current pattern you're matching), use whatever control structures you like, etc. The only restriction is that you shouldn't jump out of the *pat.match..pat.endmatch* statement, as just you were just warned against.

**Warning:** Do not write a short HLA procedure that contains a sequence of pattern matching function calls that you except to call from within a *pat.match..pat.endmatch* statement. The proper return address may not be sitting on the top of stack when you attempt to return back to the *pat.match..pat.endmatch* statement. and the usual "arrgh! The stack is messed up!" chaos will ensue. It is possible to write your own pattern matching functions, but they have to be written in a special way. There are instructions on how to do this at the end of this chapter. Although you cannot create a simple procedure in this manner, invoking macros should be okay as long as the expanded text would work properly at the point of the invocation.

## 25.5 Nesting Pattern Matching Statements

Suppose, using only the *pat.oneChar* pattern matching function, you wanted to match one of the following strings: "c", "cd", "ce", or "cde". You could solve this problem thusly:

```
pat.match( testString );

pat.oneChar( 'c' );
pat.EOS();
stdout.put( "Encountered 'c'" nl );

pat.alternate

pat.oneChar( 'c' );
pat.oneChar( 'd' );
pat.EOS();
stdout.put( "Encountered 'cd'" nl );

pat.alternate

pat.oneChar( 'c' );
pat.oneChar( 'e' );
pat.EOS();
stdout.put( "Encountered 'ce'" nl );

pat.alternate

pat.oneChar( 'c' );
pat.oneChar( 'd' );
pat.oneChar( 'e' );
pat.EOS();
stdout.put( "Encountered 'cde'" nl );

pat.if_failure

stdout.put( "Failed to match 'c', 'cd', 'ce', or 'cde'" nl );

pat.endmatch;
```

However, you'll notice that there is a bit of duplicated code here. This makes your program unnecessarily larger and slower. For example, supposed that *testString* holds the value "x". The code above will try the main pattern and all three alternates before failing. Furthermore, note that all of these patterns begin with the character 'c'. Wouldn't it be nice to factor out the test for 'c' and have only a single call to test for this character? Well, as it turns out, this is quite easy to accomplish – *pat.match..pat.endmatch* statements are nestable and recursive, so factoring out subpatterns is fairly easy.

Before discussing how to nest *pat.match* statements, we need to make a quick detour and discuss the *pat.fail* function call. This function does exactly what its name implies: if you execute it within a matching section, that section immediately fails. If you've been told (as you have) not to use control structures like IF/ELSE within the pattern matching code and that you should only use straight-line code sequences, you might wonder about the

purpose of the *pat.fail* function. After all, if some pattern matching sequence contains a call to *pat.fail*, that sequence is always going to fail even if all the functions prior to that point succeed. So why even bother executing the sequence at all? Well, although you should not execute control structures like an IF statement within a pattern matching sequence, don't forget that the *pat.match..pat.endmatch* is, essentially, an IF/ELSE statement. And, as the title of this subsection suggests, you can nest *pat.match* statements inside other *pat.match* statements. Therefore, you do have an IF statement – the *pat.match* statement. Consider the following (non-functional) first attempt at using a *pat.match* statement nested inside another to solve the problem given earlier:

```
pat.match( testString );

    pat.oneChar( 'c' );
    pat.match( ??? );

        pat.EOS();
        stdout.put( "matched 'c'" nl );

pat.alternate;

    pat.oneChar( 'd' );
    pat.match( ??? );

        pat.oneChar( 'e' );
        pat.EOS();
        stdout.put( "matched 'cde'" nl );

pat.alternate

    pat.EOS();
    stdout.put( "matched 'cd'" nl );

pat.if_failure

    stdout.put( "Failed to match 'c', 'cd', 'ce', or 'cde'" nl );

pat.endmatch;

pat.alternate

    pat.oneChar( 'e' );
    pat.EOS();
    stdout.put( "Matched 'ce'" nl );

pat.if_failure

    stdout.put( "Failed to match 'c', 'cd', 'ce', or 'cde'" nl );

pat.endmatch;

pat.if_failure

    stdout.put( "Failed to match 'c', 'cd', 'ce', or 'cde'" nl );

pat.endmatch;
```

There are two obvious problems with this code sequence. First of all, the easy one: what do we pass the second and third *pat.match* calls? We cannot pass it the original string because we need to pass it a sequence consisting of the characters after the first 'c' that we've already matched. That is, we need to pass this statement the current cursor position (which is in ESI) and the current end of sequence address (which is in EDI). Therefore, we can use the following code to achieve this:

```

pat.match( testString );

pat.oneChar( 'c' );
pat.match( esi, edi );

    pat.EOS();
    stdout.put( "matched 'c'" nl );

pat.alternate;

    pat.oneChar( 'd' );
    pat.match( esi, edi );

        pat.oneChar( 'e' );
        pat.EOS();
        stdout.put( "matched 'cde'" nl );

pat.alternate

    pat.EOS();
    stdout.put( "matched 'cd'" nl );

pat.if_failure

    stdout.put( "Failed to match 'c', 'cd', 'ce', or 'cde'" nl );

pat.endmatch;

pat.alternate

    pat.oneChar( 'e' );
    pat.EOS();
    stdout.put( "Matched 'ce'" nl );

pat.if_failure

    stdout.put( "Failed to match 'c', 'cd', 'ce', or 'cde'" nl );

pat.endmatch;

pat.if_failure

    stdout.put( "Failed to match 'c', 'cd', 'ce', or 'cde'" nl );

pat.endmatch;

```

The second problem is a bit more difficult to solve. Specifically, we still haven't properly factored out the failure cases. Notice that there are three separate failure cases, all printing the same message. We'd like to have a single failure case than handles everything. As you may have guessed, this is where the *pat.fail* function comes in.

Although you can nest *pat.match* statements, a *pat.match..pat.endmatch* statement, by itself, is not a pattern matching function. It's just a "semantic action" that should appear after all your other pattern matching function calls. However, by the judicious use of the *pat.fail* function, we can turn it into a bonafide pattern matching function. Now a call to *pat.fail* within a pattern matching section isn't going to be very interesting. That's simply going to transfer control to the *pat.match*'s associated *pat.if\_failure* section. However, what happens if we put the call to *pat.fail* inside the *pat.if\_failure* section? The *pat.if\_failure* section is not a pattern matching section. If you execute any pattern matching function inside a *pat.if\_failure* section, they will not be processed within that *pat.match..pat.endmatch* statement. Instead, they will be processed by any enclosing *pat.match..pat.endmatch* statement. The following example demonstrates how to use *pat.fail* to simplify the previous code:



```

pat.match( testString );// 1

pat.oneChar( 'c' );
pat.match( esi, edi );// 2

    pat.EOS();
    stdout.put( "matched 'c'" nl );

pat.alternate;

    pat.oneChar( 'd' );
    pat.match( esi, edi );//3

        pat.oneChar( 'e' );
        pat.EOS();
        stdout.put( "matched 'cde'" nl );

pat.alternate

    pat.EOS();
    stdout.put( "matched 'cd'" nl );

pat.if_failure

    pat.fail(); // Fails to pat.match #2's if_failure section

pat.endmatch;

pat.alternate

    pat.oneChar( 'e' );
    pat.EOS();
    stdout.put( "Matched 'ce'" nl );

pat.if_failure

    pat.fail();// Fails to pat.match #1's if_failure section

pat.endmatch;

pat.if_failure

    stdout.put( "Failed to match 'c', 'cd', 'ce', or 'cde'" nl );

pat.endmatch;

```

By the way, you would never actually want to match these four strings this way. There is a `pat.matchStr` function that provides a much better solution for this problem. Just so you don't walk away thinking these pattern matching functions are terrible, here's a better solution:

```

pat.match( testString );// 1

    pat.matchStr( "c" );
    pat.EOS();
    stdout.put( "matched c" nl );

pat.alternate

```

```

    pat.matchStr( "cd" );
    pat.EOS();
    stdout.put( "matched cd" nl );

pat.alternate

    pat.matchStr( "ce" );
    pat.EOS();
    stdout.put( "matched ce" nl );

pat.alternate

    pat.matchStr( "cde" );
    pat.EOS();
    stdout.put( "matched cde" nl );

pat.if_failure

    stdout.put( "Failed to match 'c', 'cd', 'ce', or 'cde'" nl );

pat.endmatch;

```

Obviously, this solution is a lot easier to read and understand (and more efficient, too). The previous examples are present to demonstrate nested invocations of the *pat.match* statement.

## 25.6 Cleanly Nesting Patterns

The previous section demonstrated how to nest patterns and handle the failure case by using the *pat.fail* function. In fact, there are several problems with this approach. In particular, the *pat.match..pat.endmatch* statement is not a pattern matching function (from the perspective of the *pat.match* statement), therefore, for reasons already noted and many unstated, it's not a good idea to use this statement outside the user code ("semantic action") in a pattern matching section. Fortunately, the HLA Standard Library pattern matching module provides a macro that allows you to collect a sequence of pattern matching functions and treat them as though they were a single pattern matching function: the *pat.onePat..pat.endOnePat* statement. The syntax for this statement is the following:

```

pat.onePat;

    <<sequence of pattern matching functions>>

pat.endOnePat;

```

The *pat.onePat..pat.endOnePat* statement is quite similar to the *pat.match..pat.endmatch* statement with three major differences:

There is no *pat.if\_failure* section in a *pat.onePat* statement (though *pat.alternate* sections are perfectly allowable).

You don't pass the match sequence parameter(s) to

*pat.onePat* – it uses the current cursor and end of sequence pointers.

You generally don't put any user code inside the

*pat.onePat..pat.endOnePat* sequence (you could, but it's equivalent to putting user statements in the middle of your pattern matching code).

The *pat.onePat* statement can be thought of as a parenthetical pattern matching expression. That is, it groups together a sequence of pattern matching functions and the success of *pat.onePat* depends entirely upon the success (or failure) of the group of pattern matching statements it encloses. We can use the *pat.onePat* statement to provide another example of a "clean" version of the code in the previous section:

```

static
    index:dword;

```

```

msg:string[4] :=
[
    "matched 'c'" nl,
    "matched 'cd'" nl,
    "matched 'ce'" nl,
    "matched 'cde'" nl
];

pat.match( testString );// 1

// Match the leading 'c':

pat.oneChar( 'c' );
mov( 0, index );// matched 'c'
pat.onePat;

    // See if a 'd' follows the 'c':

pat.oneChar( 'd' );

// See if an 'e' follows the 'd':

pat.onePat;

    pat.oneChar( 'e' );
    mov( 3, index );// matched 'cde'

pat.alternate

    // Note: in the absence of a pattern
    // matching function, this pattern
    // always succeeds.

    mov( 1, index );// matched 'cd'

pat.endOnePat;

pat.alternate

    // See if an 'e' follows the 'c':

pat.oneChar( 'e' );
mov( 2, index ); // matched 'ce'

pat.endOnePat;
pat.EOS();

mov( index, eax );
stdout.put( "Matched '", msg[eax*4], "'" nl );

pat.if_failure

stdout.put( "Failed to match 'c', 'cd', 'ce', or 'cde'" nl );

pat.endmatch;

```

True, this isn't quite as clean as the string example, but you cannot always convert a complex pattern to a few string compares.

Probably the most famous example of a pattern matching sequence is the following, which takes advantage of alternation and parenthetical patterns (i.e., *pat.onePat*):

```

pat.match( someString );

    pat.onePat;

        pat.matchStr( "black" );

    pat.alternate

        pat.matchStr( "blue" );

pat.endOnePat;
pat.oneChar( ' ' );
pat.onePat;

    pat.matchStr( "berry" );

pat.alternate

    pat.matchStr( "bird" );

pat.endOnePat;
stdout.put( "matched" nl );

pat.if_failure

    stdout.put( "Failed to match" nl );

pat.endmatch;

```

This example matches the string "black berry", "blue berry", "black bird", and "blue bird".

## 25.7 Backtracking

One extremely important facility that the HLA Standard Library pattern matching routines provide is *backtracking*. To understand why backtracking is important, we must expand your pattern matching function repertoire. Up to this point, you've seen *pat.oneChar* that matches exactly one character and *pat.matchStr* that matches a specific string of characters. These functions always match a fixed number of characters (one in the case of *pat.oneChar* and *n* characters, where *n* is the length of the parameter string, in the case of *pat.matchStr*). Some stdlib pattern matching functions, however, match an arbitrary number of characters. For example, consider *pat.oneOrMoreChar*; as its name implies, this function matches one or more occurrences of the same character. That is, a call such as "pat.oneOrMoreChar( 'a' );" will succeed if it can match at least one 'a' character, but it will consume as many 'a' character as it finds in the input stream. The *pat.oneOrMoreChar* eagerly matches characters. That is, it will match as many characters as it finds starting at the cursor position through to the end of the match sequence. Generally, this is desirable for a function with a name like *pat.oneOrMoreChar*, but it can lead to some problems. Consider the following example:

```

pat.match( "aaaa" );

    pat.oneOrMoreChar( 'a' );
    pat.oneChar( 'a' );
    pat.EOS();
    stdout.put( "matched a string of two or more a's" nl );

pat.if_failure

    stdout.put( "Failed to match a string of two or more a's" nl );

```

```
pat.endmatch;
```

In the absence of backtracking, this example would fail and print the message in the *pat.if\_failure* section. This would happen because the *pat.oneOrMoreChar* function would eagerly match all the characters in the match sequence (stopping at the end of the sequence) and the next call to *pat.oneChar* would fail because all of the characters have been consumed. Logically, however, this pattern match should succeed. After all, "aaa" certainly matches the *pat.oneOrMoreChar( 'a' )*; function call so there is no reason that this pattern shouldn't succeed. The call to *pat.oneOrMoreChar* should match the first three 'a' characters, the call to *pat.oneChar* should match the fourth, and then the call to *pat.EOS* should match the end of the sequence. In the presence of backtracking, this is exactly what happens.

The HLA Standard Library pattern matching functions that match a variable number of characters all support backtracking. Here's how backtracking works in the previous example:

The *pat.oneOrMoreChar* function eagerly matches as many characters as it can.

The

*pat.oneChar* attempts to match a single 'a' character. It fails. Control does not immediately transfer to the failure section, however, because the *pat.oneOrMoreChar* function has set up a backtracking frame on the stack (this is the extra stuff that pattern matching functions leave on the stack). In the presence of a backtracking frame on the stack, control transfers back inside the function that pushed the backtracking information (*pat.oneOrMoreChar* in this case).

Inside

*pat.oneOrMoreChar*, the code backs off one character position, so now it matches only "aaa" rather than "aaaa" and returns as before (still leaving a backtrack frame on the stack, in case it's needed).

Because

*pat.oneOrMoreChar* has backed up one character at the end of the string, the cursor now points at a single 'a' character, which the *pat.oneChar* function matches.

After

*pat.oneChar* matches the 'a' character, the cursor is left at the end of the string and the *pat.EOS* function call matches, so the whole statement matches the string.

One area where you can get into big trouble with backtracking is the inclusion of user code ("semantic actions") within the pattern matching code. Because backtracking will cause the reexecution of various instructions within the pattern matching sequence, you can get unexpected results if backtracking occurs. Consider the following example:

```
pat.match( "ccc" );
  pat.oneOrMoreChar( 'c' );
  stdout.put( "Matched first 'c'" nl );
  pat.oneOrMoreChar( 'c' );
  stdout.put( "Matched second 'c'" nl );
  pat.oneOrMoreChar( 'c' );
  stdout.put( "Matched third 'c'" nl );

pat.if_failure

  stdout.put( "failed" nl );

pat.endmatch;
```

This code produces the following output because of backtracking:

```
Matched first 'c'
Matched first 'c'
Matched second 'c'
Matched first 'c'
Matched second 'c'
Matched second 'c'
Matched third 'c'
```

For an explanation of this output, see the section on "Lazy / Eager Evaluation and Pattern Matching Performance" a little later in this document. What's important to realize here is that burying user statements (especially those that affect the outside world, such as output statements) is a very bad idea.

## 25.8 Pattern Components

Thus far, you've seen four different types of pattern objects: parenthetical patterns, characters, strings, and the end of sequence. The HLA Standard Library pattern matching module provides several additional pattern object types. Specifically, the patterns module provides pattern matching functions that test the following:

- Character set membership
- Characters (case sensitive)
- Characters (case insensitive)
- Strings (case sensitive)
- Strings (case insensitive)
- Words (strings delimited by special characters, case sensitive)
- Words (case insensitive)
- Whitespace
- End of string/sequence
- Arbitrary character matching
- Subpatterns
- Cursor position within a match sequence

In addition to these built-in patterns, it is possible for you to extend the pattern matching module by writing your own pattern matching functions. A later section in this document will describe how that is done.

The character and character set pattern matching functions are, by far, the most flexible and powerful of the bunch. Each of these three groups (character sets, case-sensitive characters, and case-insensitive characters) about 20 functions that let you:

- Match the character at the cursor position without advancing the cursor (`peekCset`, `peekChar`, `peekiChar`)
- Match the character at the cursor position and advance the cursor (`oneCset`, `oneChar`, `oneiChar`).
- Match an arbitrary number of characters up to the first occurrence of some character (`upToCset`, `upToChar`, `upToiChar`).
- Match zero or one characters (`zeroOrOneCset`, `zeroOrOneChar`, `zeroOrOneiChar`).
- Match zero or more characters (`zeroOrMoreCset`, `zeroOrMoreChar`, `zeroOrMoreiChar`).
- Match one or more characters (`oneOrMoreCset`, `oneOrMoreChar`, `oneOrMoreiChar`).
- Match exactly
  - $n$  characters (`firstNCset`, `exactlyNCset`, `firstNChar`, `exactlyNChar`, `firstNiChar`, `exactlyNiChar`), where  $n$  is a parameter value.
- Match
  - $n$  or fewer characters (`norLessCset`, `norLessChar`, `norLessiChar`), where  $n$  is a parameter value.
- Match
  - $n$  or more characters (`norMoreCset`, `norMoreChar`, `norMoreiChar`), where  $n$  is a parameter value.
- Match between
  - $n$  and  $m$  characters (`ntoMCset`, `exactlyNtoMCset`, `ntoMChar`, `exactlyNtoMChar`, `ntoMiChar`, `exactlyNtoMiChar`), where  $n$  and  $m$  are parameter values.

There are also lazy versions of many of the functions in the above list. We'll discuss the lazy functions in the next section on Eager and Lazy evaluation. As for the specifics of these functions, we'll discuss them in the reference section later in this document.

The important thing to note is that many of these pattern matching functions match an arbitrary or parameterized number of characters. For example, a call like the following:

```
pat.exactlyNCset( { 'a', 'b', 'c' }, 5 );
```

matches exactly five characters and all of them must be members of the set {'a', 'b', 'c'}. The functions that begin with "zeroOrOne..." will either match a single character, or they will succeed without advancing the cursor. The "zeroOrMore..." functions will match as many copies of the character as they can, or they will

succeed without matching any characters. The "oneOrMore..." functions must match at least one character, but will happily match any number of characters afterwards, as well. The "firstN..." functions will match exactly  $n$  copies of the specified character (set); the "exactlyN..." functions also match exactly  $n$  characters, but they differ from the "firstN..." functions insofar as the "firstN..." functions don't care what character (if any) appears in the  $n+1^{\text{st}}$  position. The "exactlyN..." functions require the  $n+1^{\text{st}}$  character to either be nonexistent (i.e., there were only  $n$  characters in the string) or it must not be the character (or in the character set) that the function matches. The "norLess..." functions match between zero and  $n$  copies of a character. The "norMore..." functions match, you guessed it,  $n$  or more characters in the string. The "nToM..." and "exactlyNtoM..." functions match between  $n$  and  $m$  copies of the character in the match sequence; the difference between the two is that the "ntoM..." functions allow the  $m+1^{\text{st}}$  character to match the pattern whereas the "exactlyNtoM..." functions fail if the  $m+1^{\text{st}}$  character matches. With all of these functions, it's pretty easy to concoct some pattern matching sequence that can match just about anything.

Though there aren't quite as many string matching functions as there are character and character set functions, there are still a useful variety of functions available. You can match a string (as you've already seen) with the `pat.matchStr` function. There's a corresponding `pat.matchiStr` function that does a case insensitive comparison. You can also match all the characters up to (and including) a string with the `pat.upToStr` function (`pat.upToiStr` is the case-insensitive version); `pat.matchToStr` and `pat.matchToiStr` are similar except they match all characters up to, but not including, the string you pass as a parameter.

There are several other string matching functions you'll want to use. Please consult the reference section at the end of this document for more details on those (especially the whitespace matching functions).

## 25.9 Lazy / Eager Evaluation and Pattern Matching Performance

Although backtracking is an incredibly useful feature to have, in some very degenerate cases backtracking can produce very slow results. Consider the following example:

```
pat.match( "aaaaaa" )

    pat.zeroOrMoreChar( 'a' );
    pat.oneOrMoreChar( 'a' );
    pat.oneOrMoreChar( 'a' );
    pat.oneOrMoreChar( 'a' );
    pat.oneOrMoreChar( 'a' );
    pat.oneOrMoreChar( 'a' );
    pat.oneOrMoreChar( 'a' );
    stdout.put( "succeeded" nl );

pat.if_failure

    stdout.put( "failed" nl );

pat.endmatch;
```

Now this particular pattern will succeed. It does so by having the first function match zero characters and all the remaining functions match a single character. This looks simple enough, but if you look closely, you discover that it takes a huge amount of CPU time to match this string. Let's consider what happens here:

The call to `pat.zeroOrMoreChar` eagerly matches the entire string.

The first call to `pat.oneOrMoreChar` fails because the first call has consumed all the characters. So backtracking occurs and `zeroOrMoreChar` releases one character, which the first call to `oneOrMoreChar` succeeds in matching (this is the second call to that function, by the way).

Control transfers to the second `pat.oneOrMoreChar` function. It fails because the previous two functions have consumed all the characters in the string. So back tracking occurs. The second call to `oneOrMoreChar` backtracks to the first call, which tries to give up a character. But when it does, it fails to match, so it back tracks back up to the `zeroOrMoreChar` call, which backs up a second character and control transfers back to the first `oneOrMoreChar` call, with the string "aa". The first `oneOrMoreChar` call matches

*both* of these characters, so when the call to the second `oneOrMoreChar` takes place, it fails again. Once again backtracking occurs, this time, however, the first `oneOrMoreChar` call can give up one character and still

succeed. So control flows back to the second `oneOrMoreChar` call and it succeeds. Then control falls through to the third `oneOrMoreChar` call and it fails, and the process starts all over again. To make a (very) long story short, backtracking is going to have exponential worst-case time complexity (that is, it will take on the order of  $2^n$  operations to perform the character match operation).

Though such degenerate cases rarely occur in practice, eager evaluation can be quite expensive when such conditions arise. The solution to this particular problem is to use *lazy evaluation* rather than eager evaluation. For all the functions that match an arbitrary number of characters, there is usually a complementary function that begins with `l_` that performs the same test using lazy evaluation. In the example above, the complementary functions are `pat.l_zeroOrMoreChar` and `pat.l_oneOrMoreChar`. The difference between the eager and the lazy functions is that the eager functions will attempt to match as many characters as the possibly can when first called, and will back off only when backtracking occurs. Lazy functions, on the other hand, will match as few characters as possible and will match more characters only when backtracking occurs. Consider the following rework of the previous example:

```
pat.match( "aaaaaa" )

pat.l_zeroOrMoreChar( 'a' );
pat.l_oneOrMoreChar( 'a' );
pat.l_oneOrMoreChar( 'a' );
pat.l_oneOrMoreChar( 'a' );
pat.l_oneOrMoreChar( 'a' );
pat.l_oneOrMoreChar( 'a' );
pat.l_oneOrMoreChar( 'a' );
pat.l_oneOrMoreChar( 'a' );
stdout.put( "succeeded" nl );

pat.if_failure

stdout.put( "failed" nl );

pat.endmatch;
```

This function will succeed, just as before, but it won't consume much CPU time at all. The first call matches the minimum number of characters (zero), the remaining functions also match the minimum number of characters (one each), so this code matches the string in one pass without any backtracking.

Lazy evaluation does not completely solve the problem. It is perfectly possible to create a degenerate string that causes lazy evaluation to require exponential time complexity (i.e., run very slow). Indeed, eager evaluation is probably best as the default case. Nonetheless, if you have a good idea of what your match sequences (input strings) will be like, then you can choose eager or lazy evaluation as appropriate to produce the best performance.

In the absence of user code ("semantic actions"), lazy and eager evaluation always produce the same result (even if the performance characteristics are different). That is, if one pattern using eager evaluation matches, then the comparable pattern using lazy evaluation will also match. However, once you embed user statements between the pattern matching functions, the recurring execution of those statements can be greatly affected by your choice of lazy versus eager evaluation. One more reason to avoid, as much as possible, embedding user instructions in the pattern matching sequences.

Another way to view eager versus lazy evaluation is that eager evaluation always attempts a *maximal match* (matching as many characters as possible) whereas lazy evaluation does a *minimal match* (matching as few characters as possible). In the absence of backtracking, the two approaches could match entirely different strings; but with backtracking present, either method will match a string (though the way they match, and the execution of the associated semantic actions, might be different). If lazy and eager evaluation techniques match a string by matching different substrings during the matching process (that is, if there are two or more ways the code can match the string), we say that the matching operation is *ambiguous*.

## 25.10 Regular Expressions

If you are familiar with regular expression syntax (e.g., from Unix shell interpreters, various editors, or programs like *grep*), you may find the HLA Standard Library pattern matching routines easier to understand if they are explained in terms of a regular expression syntax. This section will draw some parallels between the HLA Standard Library pattern matching functions and the typical syntaxes that regular expressions use.



In a simple regular expression language, there are two types of characters: *metasymbols* and *alphabetic characters*. Metasymbols have special meaning to the regular expression language and typically include symbols such as '\*', '+', '?', '.', '(', ')', and '|'. Alphabetic characters are symbols from a predefined alphabet (an alphabet is simply a set of characters, it isn't necessarily the characters 'a'..'z' from the English alphabet). In most computer systems, the alphabetic is the set of ASCII or UNICODE (UTF-8) characters, sans the metasymbols. For the HLA Standard Library, the alphabet is the set of all 7-bit ASCII characters except the NUL character (ASCII code 0).

In a typical regular expression language (e.g., *grep*'s regular expression language), the metasymbols are typically:

. ? \* + | ( ) [ ] ^ \ ' "

The alphabet is the set of all other characters in the system's native character set (e.g., 7-bit ASCII characters). In the event you want to specify one of the metasymbols (which are valid ASCII characters) as standard characters in the alphabet rather than as metasymbols, you can *escape* the meaning of the symbol by prefacing it with a '\' character. For example, the character sequence '\\*' represents a single asterisk character, '\(' represents a single left parenthesis character, and '\\' represents a single backslash character. When a character has an escape prefix on it, it is treated as any other character in the alphabet.

We can define a regular expression with the following rules:

If *a* is any single character from the alphabet (or an escaped character), then *a* is a regular expression and it matches the single character *a*<sup>1</sup>.

If *a* is any single character from the alphabet, then '*a*' is a regular expression and it matches the single character *a*. In many regular expression languages, *a* can actually be a metacharacter (other than ') and quoting the character also escapes it.

If *b* is sequence of zero or more characters from the alphabet, then "*b*" is a regular expression and it matches the string *b*.

The '.' metasybol represents any character in the character set and is a regular expression. Note the difference between '.' and

*a* from the previous rule. The *a* represents any single character from the character set whereas '.' is the actual period character. The regular expression *a* matches only the character represented by *a*, the regular expression represented by '.' will match any character in the alphabet.

If *r* is a regular expression and *s* is a regular expression, then the concatenation of *r*+*s* is also a regular expression and it matches the sequence of characters matched by *r* immediately followed by the sequence of characters matched by *s*. In regular expression terms, this is generally written as *rs*. Note that we may apply this rule recursively to generate strings of any length to match. For example, the string "hello" can be generated as follows:

regex = rs (by definition)

rs = rss (by substituting rs, a regular expression for r)

rss = rsss (by substituting rs for r).

rsss = rssss (by substituting rs for r).

rssss = hello (by substituting 'h' for r, and 'ello' for each of the regular subexpressions ssss, respectively).

If *r* is a regular expression, then *r*? is also a regular expression and it represents zero or one occurrences of *r* (that is, it optionally matches *r*).

If *r* is a regular expression, then *r*\* is a regular expression and it matches zero or more concatenated occurrences of *r*. Note that *r* can be any regular expression, not just a single character. For example, the regular expression '.'\*' matches zero or more characters from the alphabet whereas 't\*' only matches zero or more 't' characters.

If *r* is a regular expression, then *r*+ is also a regular expression and matches one or more instances of the regular expression *r*. This is actually a shorthand notation for *rr*\* (that is, one instance of *r* followed by zero or more instances of *r*).

If *r* and *s* are regular expressions, then *r*|*s* is also a regular expression and it will match exactly one occurrence of *r* or *s* (alternation).

---

1. Technically speaking, regular expressions generate strings rather than recognize strings. However, from the theory of computation we can easily show that generation and recognition are equivalent operations, so as this document discusses pattern matching we'll use the term "recognize" or "match" when discussing the behavior of a regular expression.

If  $r$  is a regular expression, then  $(r)$  is also a regular expression and it matches the same strings that  $r$  matches. As for arithmetic expressions, parenthesis are normally used to override precedence and group expressions.

$[charset]$  is a regular expression and matches exactly one character from the specified character set. Character sets have the following definition:

A single character  $a$ , from the alphabet, is a legal character set and the character set  $[a]$  matches this single character.

A character set of the form

$[a-b]$ , where  $a$  and  $b$  are both characters in the alphabet with  $a$ 's ordinal value being less than or equal to  $b$ 's ordinal value, is a character set and will match a single character whose value is between  $a$ 's and  $b$ 's ordinal values (inclusive).

If  $[c]$  and  $[d]$  are valid character set formulations from items (1) and (2) above, then  $[cd]$  is a valid character set and matches any character in the union of the two sets  $c$  and  $d$ . For example,  $[a-zA-Z]$  is the union of  $[a-z]$  and  $[A-Z]$  and represents the set of all (ASCII/English) alphabetic characters.

If  $[c]$  is a valid character set, then  $[^c]$  is also a valid character set and represents the complement of the character set  $c$ . For example,  $[^a-zA-Z]$  represents the set of all non-alphabetic characters (in the ASCII character set, anyway). Note that the  $^$  symbol must appear immediately after the  $[$  and this is the only place that the  $^$  symbol has special meaning.

These few rules are (more than) sufficient to define all regular expressions. Sometimes, however, it is convenient to define a few extra rules to make it easy to specify some complex patterns. In some regular expression languages, for example, an expression of the form  $r:[n]$ , where  $r$  is a regular expression and  $n$  is an integer value, will match exactly  $n$  occurrences of the regular expression  $r$ . A regular expression of the form  $r:[n,m]$ , where  $r$  is some regular expression and  $n$  and  $m$  are integer values with  $n \leq m$  will match between  $n$  and  $m$  occurrences of the regular expression  $r$ .

Here are some common regular expressions and the strings they match:

$[a-zA-Z\_][a-zA-Z\_0-9]^*$	HLA identifier
$[0-9]^+$	Unsigned integer constant
$[0-9]([\_0-9]^*[0-9])^?$	HLA unsigned integer
$(\+ -)^?[0-9]^+$	Signed integer constant
$[+ -]^?[0-9]^+(\.[0-9]^*)^?([eE](\+ -)^?[0-9]^+)^?$	Real constant
if	HLA reserved word "if"

Though it would certainly be possible to write some HLA macro that processes regular expressions using the standard syntax given above (for *grep*-like regular expressions), most pattern-matching operations in the HLA Standard Library pattern matching module are accomplished using function calls. This is a bit more typing (and a bit more text to read), but the result is easier to read and understand than a cryptic regular expression, particularly if the regular expression is complex. Of course, the other main difference is that HLA's syntax allows the incorporation of semantic actions (user code to execute on a match), something that traditional regular expression languages do not provide.

It should go without saying, given the number of functions present in the HLA Standard Library pattern matching module, that the `stdlib` provides a rich set of functions that allow you to process any type of regular expression that you can express using a *grep*-like notation. Let's cover the conversion of *grep*-like regular expressions to HLA Standard Library pattern matching code.

If  $a$  represents a single character (either a character literal constant or a character variable in HLA), then `pat.oneChar( $a$ )`; will succeed if the character at the cursor position matches  $a$ , it will fail otherwise. Note that other than `'` and `"`, HLA does not have any metacharacters. You either supply a character variable or a character constant as the `pat.oneChar` operand.

If  $b$  represents a string of characters (either a string variable or an HLA literal or manifest string constant) then `pat.matchStr( $b$ )`; will succeed if the character sequence at the cursor matches the character string  $b$ . It fails otherwise.

The HLA `stdlib` pattern matching module provides several ways to match an arbitrary character. The standard way is with the `pat.skip( $n$ )` function, where  $n$  is an unsigned integer. This function succeeds if there are at least  $n$  characters left in the match sequence string starting at the cursor position. It fails if there are fewer than  $n$  characters left in the string. To match a single arbitrary character, you would simply supply the value one as the function's argument: `pat.skip(1)`; You could also take the complete of the empty set (which is the entire character set) and pass that to the `pat.oneCset` function: `pat.oneCset(-{ })`; Note, however, that the `pat.skip` function call is more efficient.

Concatenation of two regular expressions is handled by making sequential function calls to the corresponding functions that implement the sub-regular expressions. For example, if function `pat.RRRRR` implements regular expression  $r$  and function `pat.SSSSS` implements regular expression  $s$ , then the following statements implement the regular expression  $rs$ :

```
pat.RRRRR( ... );
pat.SSSSS( ... );
```

Note that you do not have to build string matches up from individual character matches. Just use the `pat.matchStr` function when matching a sequence of (known) characters.

To match zero or more occurrences of a generic regular expression *r*, the HLA stdlib pattern matching module provides the `pat.zeroOrMorePat...pat.endZeroOrMorePat` statement. You place the statement(s) that implement the regular expression *r* in the body of the `pat.zeroOrMorePat...pat.endZeroOrMorePat` statement and the pattern matching code will attempt to match zero or more occurrences (note that such regular expressions always succeed, as matching zero occurrences is legal).

The HLA stdlib pattern matching module also provides several special case functions that will match zero or more occurrences of:

- Any single character (case sensitive or case insensitive)
- Any character from a character set
- Any white space character

Using these special functions is far more efficient than using the `pat.zeroOrMorePat...pat.endZeroOrMorePat` statement, so you should call these functions if appropriate. For example, to match zero or more alphabetic characters, you'd probably want to use the built-in `pat.zeroOrMoreCset` function thusly:

```
pat.zeroOrMoreCset( { 'a'..'z', 'A'..'Z' } );
```

To match one or more occurrences of a generic regular expression *r*, the HLA stdlib pattern matching module provides the `pat.oneOrMorePat...pat.endOneOrMorePat` statement. You place the statement(s) that implement the regular expression *r* in the body of the `pat.oneOrMorePat...pat.endOneOrMorePat` statement and the pattern matching code will attempt to match one or more occurrences. The statement fails if there is not at least one occurrence of the regular expression

The HLA stdlib pattern matching module also provides several special case functions that will match one or more occurrences of:

- Any single character (case sensitive or case insensitive)
- Any character from a character set
- Any white space character

Using these special functions is far more efficient than using the `pat.oneOrMorePat...pat.endOneOrMorePat` statement, so you should call these functions if appropriate. For example, to match an integer value consisting of one or more decimal digits, you'd probably want to use the built-in `pat.oneOrMorePat` function thusly:

```
pat.oneOrMoreCset( { '0'..'9' } );
```

Alternation is handled by the HLA stdlib pattern matching `pat.alternate` statement. For simple regular expressions where the alternation occurs at the outermost level (that is, having the lowest precedence in the regular expression) you can simply use the `pat.alternate` statement within the outermost `pat.match...pat.endmatch` statement. For more complex regular expressions, when the alternation appears inside parenthetical expressions, your best bet is to use the `pat.onePat...pat.alternate...pat.endOnePat` statement to achieve the alternation. the earlier (black|blue)(berry|bird) regular expression example comes to mind here:

```
pat.match( someString );

pat.onePat;

    pat.matchStr( "black" );

pat.alternate

    pat.matchStr( "blue" );

pat.endOnePat;
pat.oneChar( ' ' );
pat.onePat;
```

```

        pat.matchStr( "berry" );

    pat.alternate

        pat.matchStr( "bird" );

    pat.endOnePat;
    stdout.put( "matched" nl );

pat.if_failure

    stdout.put( "Failed to match" nl );

pat.endmatch;

```

Paranthetical regular expressions are handled by the *pat.onePat..pat.endOnePat* statement in HLA's pattern matching module. The statements inside this block are executed with higher precedence than the outside code. Consider the following regular expression that matches "blackbird", "bluebird", or "canary":

```
canary | (black|blue) bird
```

Had this been written as "canary | black | blue bird" it wouldn't match the correct strings (it would match "canary", "black", or "blue bird"). Parentheses adjust the precedence of the expression ( "|" normally has the lowest precedence of all the regular expression operators, concatenation has very high precedence) to give us the expression we want. To implement the correct regular expression in HLA code, we use the *pat.onePat* and *pat.endOnePat* as our parentheses around the subexpressions:

```

pat.match( someString );

    pat.matchStr( "canary" );

pat.alternate

    pat.onePat

        pat.matchStr( "black" );

    pat.alternate

        pat.matchStr( "blue" );

    pat.endOnePat;
    pat.matchStr( "bird" );

pat.endmatch;

```

The HLA language provides character sets as a built-in data type, so if you want to match a character set you simply call one of the *pat.\*Cset* function and pass a character set as the function's argument. If you want to match against the complement of a character set, you can take the complement by using the set negation ('-') operator, e.g.,

```

// Match non-alpha chars

pat.zeroOrMoreCset( -{ 'a'...'z', 'A'...'Z' } );

```

See the function reference for a complete description of all the HLA pattern matching functions.

## 25.11 Pattern Matching Statements

The HLA Standard Library pattern matching module basically breaks up the pattern matching operations into two different categories: statements and functions. Statements are always implemented as macros, functions might be macros or HLA procedures. This section will describe the statements, the next section will describe all the pattern matching functions.

### **pat.match and pat.endmatch Syntax**

The HLA *pat.match* and *pat.endmatch* macros provide the basic tools for pattern matching. These macro statement allow one of the following two syntaxes:

```
// Match syntax #1:

pat.match( StringValue );
  << Sequence of match functions>>
  << Code to execute on a successful match >>

  pat.if_failure

  << Code to execute if the match fails >>

pat.endmatch;
```

StringValue is either an HLA string variable or a string constant.

```
// Match syntax #2:

pat.match( StartOfStr, EndOfStr );
  << Sequence of match functions>>
  << Code to execute on a successful match >>

  pat.if_failure

  << Code to execute if the match fails >>

pat.endmatch;
```

The *StartOfStr* and *EndOfStr* parameters (in syntax #2) must be dword pointers to characters. *StartOfStr* points at the first character of a sequence of characters to match against. *EndOfStr* must point at the first byte *beyond* the last character in the sequence to consider.

The *pat.match* statement, along with many of the matching functions, pushes data onto the stack that may not be cleaned up until execution of the *pat.endmatch* statement. Therefore, you must never jump into a *pat.match..pat.endmatch* block. Likewise, unless you are prepared to clean up the stack yourself, you should not jump out of a *pat.match..pat.endmatch* block<sup>2</sup>.

During a normal match operation, the *pat.match* block executes the sequence of string matching functions. If all the functions in the list execute and successfully match their portion of the string, control falls through to the statements after the match sequence. This code should do whatever is necessary if the pattern matches.

On the other hand, if a failure occurs and the pattern matching routines cannot match the specified string, then control transfers to the *pat.if\_failure* section and the associated statements execute. Like an IF..THEN..ELSE statement, the program automatically jumps over the *pat.if\_failure* section if the "successful match" statements execute.

Consider the following example that matches a string containing a single HLA identifier:

```
pat.match( StrToTest );
```

---

2. If an exception occurs, the exception handling code will clean up the stack, so exceptions are a legitimate way to prematurely leave a *pat.match..pat.endmatch* block.

```

pat.oneCset( { 'a'..'z', 'A'..'Z', '_' } );
pat.zeroOrMoreCset( { 'a'..'z', 'A'..'Z', '0'..'9', '_' } );
pat.EOS;

stdout.put( "The string is a valid HLA identifier" nl );

pat.if_failure

    stdout.put( "The string is not a valid HLA id" nl );

pat.endmatch;

```

The *pat.oneCset* function matches a single character in *StrToTest* that is a member of the character set appearing in the parameter list. This call requires that the first character of *StrToTest* be an alphabetic character or an underscore.

After *pat.oneCset* matches a character, the pattern matching routines advance a *cursor* into *StrToTest* so that it points just beyond the character matched by *pat.oneCset*. Indeed, all pattern matching routines operate in this manner, they maintain a cursor (in ESI) that points beyond the characters just matched. So had *StrToTest* contained the string "Hello", ESI would be pointing at the "e" in "Hello" immediately after the execution of the *pat.oneCset* pattern matching routine.

The HLA pattern matching routines also return EBX pointing at the first character matched by the routine. In the current example being considered, EBX would be returned pointing at the "H" in "Hello" by the *pat.oneCset* routine.

The *pat.zeroOrMoreCset* routine continues where *pat.oneCset* leaves off. It matches zero or more characters (starting at the location pointed at by ESI). In this particular example, *pat.zeroOrMoreCset* matches zero or more alphanumeric and underscore characters, hence the code will match "ello" in "Hello".

The *pat.EOS* macro matches the end of the string, just to make sure there aren't any other illegal (nonalphanumeric) characters in the string. Note that *pat.zeroOrMoreCset* stops upon encountering the first non-alphanumeric character. The remainder of the pattern (EOS, in this case) must verify that *pat.zeroOrMoreCset* didn't stop on an illegal character.

Had the *StrToTest* variable contained the string "Hello", then the pattern would successfully match the string and the program would print "The string is a valid HLA identifier" and continue execution after the *pat.endmatch* statement.

Because of the way HLA pattern matching routines implement backtracking, each matching routine may leave data on the stack when it successfully returns. This information is necessary to implement backtracking. Although the *pat.endmatch* code cleans up the stack upon exit, it is important to realize that stack is not static. In particular, you cannot push data on the stack before one pattern matching routine and expect to pop it off the stack when that matching routine returns. Instead, you'll pop the data that the matching routine left on the stack (which will probably crash the system if backtracking occurs). It is okay to manipulate the stack in the code section following all the matching functions (or in the failure section), but you must leave the stack intact between calls to pattern matching routines<sup>3</sup>.

## 25.12 Alternation

Another way to handle failure is with the *pat.alternate* macro. A *pat.match..pat.endmatch* macro invocation may optionally contain one or more *pat.alternate* sections before the (required) *pat.if\_failure* section. The *pat.alternate* sections "intercept" failures from the previous section(s) and allow an attempt to rematch the string with a different pattern (somewhat like the ELSEIF clause of an IF..THEN..ELSEIF..ELSE..THEN statement). The following example demonstrates how you could use this:

```

pat.match( StrToTest );

pat.oneCset( { 'a'..'z', 'A'..'Z', '_' } );
pat.zeroOrMoreCset( { 'a'..'z', 'A'..'Z', '0'..'9', '_' } );
pat.EOS;

```

---

3. Note that it is okay to push data onto the stack, do some calculations, and then pop that data off the stack between calls to the pattern matching routines. However, you must ensure that the stack is unchanged since the last pattern matching routine (or since *pat.match*) or the pattern matching routines will malfunction.

```

    stdout.put( "The string is a valid HLA identifier" nl );

pat.alternate

    pat.oneOrMoreCset( { '0'..'9', '_' } );
    pat.EOS;

    stdout.puta
    (
        "The string is a valid HLA unsigned integer constant" nl
    );

pat.if_failure

    stdout.put
    (
        "The string is not a valid HLA id or integer constant" nl
    );

pat.endmatch;

```

In this example, if the pattern fails to match an HLA identifier, the pattern matching code attempts to see if it matches an integer constant (in the *pat.alternate* section). If this fails as well, then the whole pattern fails to match.

## 25.13 Pattern Matching Macros

The HLA patterns library implements several of the pattern matching routines as keyword macros within the *pat.match* macro. These include *pat.EOS*, *pat.position*, *pat.atPos*, *pat.skip*, *pat.getPos*, *pat.fail*, *pat.fence*, *pat.zeroOrOnePat*, *pat.zeroOrMorePat*, and *pat.oneOrMorePat*. The following sections describe each of these functions.

### **pat.EOS**

```

pat.match
    << pattern matching statements >>
    pat.EOS;
    // Note that it doesn't make sense to have any more pattern
    // matching statements here because they would never match
    // anything.
pat.endMatch;

```

The *pat.EOS* macro matches the end of the string. It succeeds if the current "cursor" value (ESI) is pointing at the end of the string to match. It fails otherwise. This macro is great for forcing a string match to consume an entire string. Specifically, by placing a *pat.EOS* macro invocation at the end of a sequence of pattern matching function calls, you cause the current pattern match to succeed only if the pattern matches the entire string.

### **pat.position( n )**

```

pat.match
    << pattern matching statements >>
    pat.position( 5 ); // Set cursor position to 5, succeed if
                       // match string is at least 5 chars long.
    << pattern matching statements >>
pat.endMatch;

```

This function repositions the cursor to character *n* in the string that *pat.match* is processing. This function fails if repositioning the cursor would move it outside the bounds of the string. Note that the index of the first character in the string is zero. The macro is great when you need to match a subpattern that begins at some fixed character position within the string.

**pat.atPos( n )**

```

pat.match
  << pattern matching statements >>
  pat.atPos( 5 );// Succeeds if above matches five characters.
  << pattern matching statements >>
pat.endMatch;

```

This function succeeds if the cursor is currently at position *n* in the string that *pat.match* is processing. It fails otherwise. This statement is useful when you need to verify that a recursive pattern doesn't exceed some bound in the string.

**pat.skip( n )**

```

pat.match
  << pattern matching statements >>
  pat.skip( 5 );// Succeeds at least five chars left in
                  // match string and advances cursor by
                  // five positions.
  << pattern matching statements >>
pat.endMatch;

```

This function advances the cursor *n* positions from its current location. This function succeeds if the new cursor position is within the bounds of the string; it fails otherwise. This function is comparable to matching a specific number of characters in the string. However, this function is much faster than *pat.arb* or one of the character set matching functions.

**pat.getPos( var dest:dword )**

```

pat.match
  << pattern matching statements >>
  pat.getPos( i );// Succeeds and puts current cursor position
                  // into 'i' variable.
  << pattern matching statements >>
pat.endMatch;

```

This function places the current cursor position in the specified destination operand. This function always succeeds. It does not affect the cursor position. This function stores zero into the *dest* variable if the cursor is at the beginning of the string.

**pat.fail**

```

pat.match
  << pattern matching statements >>
  pat.onePat
    << pattern matching statements >>
    pat.alternate
      << pattern matching statements >>
      pat.fail;//Always fails if we get to this point.
    pat.endOnePat;
  << pattern matching statements >>

  pat.alternate
    << pattern matching statements >>
pat.endMatch;

```

This forces an immediate failure, backtracking if necessary. This macro is useful for handling exceptional conditions that shouldn't match. That is, if you've matched to some point in the string and you don't want the whole pattern to succeed, executing *pat.fail* will force an immediate failure. Obviously, this macro invocation only makes sense if alternation is being used in the pattern.

**pat.fence**

```

pat.match
  << pattern matching statements >>

```



```

pat.fence; // Don't backtrack into previous statements
pat.onePat
    << pattern matching statements >>
pat.endOnePat;
<< pattern matching statements >>

    pat.alternate
    << pattern matching statements >>
pat.endMatch;

```

This function cleans all the backtracking information off the stack. Any pattern matching function following fence will not be able to backtrack to the routines immediately preceding fence in the current *pat.match* statement.

#### **pat.onePat;**

```

pat.onePat;
    << pattern matching statements >>
pat.endOnePat;

```

*<< pattern matching statements >>* are some statements that correspond to an HLA pattern sequence (it may contain pattern matching function calls, x86 code, and *pat.alternate* sections; it may not contain a *pat.if\_failure* section or a *pat.fence* invocation). The program evaluates the pattern. If it succeeds, control falls to the next statement following the *pat.pattern* call. If it fails, then control transfers directly to the *pat.if\_failure* section in the surrounding *pat.match* call.

This macro is primarily used to create "parenthetical patterns" as a convenience when creating complex patterns. Here's an example:

```

pat.match( SomeString );

pat.onePat

    pat.matchStr( "Black" );

    pat.alternate

        pat.matchStr( "Blue" );

pat.endOnePat;

pat.onePat;

    pat.matchStr( "bird" );

    pat.alternate

        pat.matchStr( "berry" );

pat.endOnePat;

stdout.put
(
    "It was 'blackbird', 'bluebird', 'blackberry', or 'blueberry'",
    nl
);

pat.if_failure

    stdout.put( "Failed to match the pattern" nl );

pat.endmatch;

```

Immediately after the *pat.endOnePat* statement, EBX points at the start of the text associated with the pattern match between the *pat.onePat* and *pat.endOnePat* calls. Therefore, you can call functions like *pat.extract* to extract the entire string matched by the pattern between the *pat.onePat* and *pat.endOnePat* calls. This function fully supports backtracking, even across the patterns within the parenthetical pattern expression.

#### **pat.zeroOrOnePat;**

```
pat.zeroOrOnePat;
<< pattern matching statements >>
pat.endZeroOrOnePat;
```

<< *pattern matching statements* >> are some statements that correspond to an HLA pattern sequence (it may contain pattern matching function calls, x86 code, and *pat.alternate* sections; it may not contain a *pat.if\_failure* section or a *pat.fence* invocation). This call invokes the pattern matching function zero or one times to match additional characters in the current string. This function always succeeds since it can match zero times. This function fully supports backtracking.

#### **pat.zeroOrMorePat;**

```
pat.zeroOrMorePat;
<< pattern matching statements >>
pat.endZeroOrMorePat
```

*Pattern* is sequence of pattern matching function calls (just like *pat.pattern* above; including allowing a *pat.alternate* section but not allowing a *pat.if\_failure* section). This call invokes the pattern matching function zero or more times to match additional characters in the current string.

#### **pat.oneOrMorePat**

```
pat.oneOrMorePat
<< pattern matching statements >>
pat.endOneOrMorePat
```

<< *pattern matching statements* >> are some statements that correspond to an HLA pattern sequence (it may contain pattern matching function calls, x86 code, and *pat.alternate* sections; it may not contain a *pat.if\_failure* section or a *pat.fence* invocation). This call invokes the pattern matching function one or more times to match additional characters in the current string. It must match at least one occurrence of the pattern in order to succeed.

## 25.14 Character Set Matching Functions

The following sections describe each of the character set matching functions provided by the HLA patterns module. These functions take (at the minimum) a character set object. The characters in the match string (at the cursor position) are tested against the characters in the set. These functions succeed if, as appropriate for the specific function, they match characters in the parameterized character set.

You'll notice that there are no "not in character set" type functions in this set. You can easily test to see if a string does not match any characters in a given character set by using the negation of the character set.

#### **procedure pat.peekCset( cst:cset );**

```
pat.match
<< pattern matching statements >>
pat.peekCset( {'0'..'9'} );           // Matches, but does not consume
                                     // a numeric character.
<< pattern matching statements >>
pat.endMatch
```

Succeeds if the following character is in the specified set. Fails otherwise. This function does not affect the cursor position of the match.

#### **procedure pat.oneCset( cst:cset );**

```
pat.match
<< pattern matching statements >>
pat.oneCset( {'0'..'9'} );           // Matches and consumes a numeric char
<< pattern matching statements >>
pat.endMatch
```

Succeeds, and advances the cursor by one position, if the character at the cursor position is in *cst*. Fails otherwise. If this function fails, it does not affect the cursor position.

The following example succeeds and advances the cursor by one position if the character at the current cursor position is an alphabetic character:

```
pat.oneCset( { 'a'..'z', 'A'..'Z' } );
```

The following example succeeds and advances the cursor if the current character is *not* an alphabetic character:

```
pat.oneCset( -{ 'a'..'z', 'A'..'Z' } ); // Note: "-" operator negates cset.
```

**procedure pat.upToCset( cst:cset );**

```
pat.match
<< pattern matching statements >>
pat.upToCset( { '0'..'9' } ); // Match all chars up to a numeric char
<< pattern matching statements >>
pat.endMatch;
```

Advances the cursor until it finds a character in *cst*. Fails if none of the characters following the cursor position (to the end of the string) are in *cst*. This advances the cursor position to the character found in the *cst* parameter. Therefore, the next matching function will begin with the character that was present in the set. Note that this function succeeds and skips zero characters if the cursor was pointing at a character in *cst* when this function was called.

This function is great for skipping over some arbitrary number of characters until a character in the given character set is found. If you call the *pat.extract* function immediately after this function call, you'll retrieve the characters skipped over by this function.

```
pat.match
<< pattern matching statements >>
pat.upToCset( { '0'..'9' } ); // Match all chars up to a numeric char
pat.extract( s ); // Extract matched chars to 's' string
<< pattern matching statements >>
pat.endMatch;
```

**procedure pat.zeroOrOneCset( cst:cset )**

```
pat.match
<< pattern matching statements >>
pat.zeroOrOneCset( { '0'..'9' } );
<< pattern matching statements >>
pat.endMatch;
```

Optionally matches a single character in the string. If the following character is in the character set, this routine advances the cursor and signals success. If the following character is not in the string, this routine simply signals success without advancing the cursor.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match the character before returning. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. If the following match routine still fails, then this routine fails.

**procedure pat.l\_ZeroOrOneCset( cst:cset )**

```
pat.match
<< pattern matching statements >>
pat.l_ZeroOrOneCset( { '0'..'9' } );
```

```
<< pattern matching statements >>
pat.endMatch;
```

Optionally matches a single character in the string. If the following character is in the character set, this routine advances the cursor and signals success. If the following character is not in the string, this routine simply signals success without advancing the cursor.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will start by matching zero characters in the string. If doing so would cause a following match routine to fail, this routine will backtrack and match one character (if possible) and then retry the following match routine. If the following routine still fails, then this routine signals failure.

```
procedure pat.zeroOrMoreCset( cst:cset );
```

```
pat.match
<< pattern matching statements >>
pat.zeroOrMoreCset( {'0'..'9'} );
<< pattern matching statements >>
pat.endMatch;
```

Matches zero or more characters in the specified character set. Because this function can match zero characters, it will always succeed. It advances the cursor beyond all the characters that it successfully matches.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up beyond the original cursor position (in which case this routine backtracks to previous functions) or the following match routine(s) succeed.

```
procedure pat.l_ZeroOrMoreCset( cst:cset );
```

```
pat.match
<< pattern matching statements >>
pat.l_ZeroOrMoreCset( {'0'..'9'} );
<< pattern matching statements >>
pat.endMatch;
```

Matches zero or more characters in the specified character set. Because this function can match zero characters, it will always succeed. It advances the cursor beyond all the characters that it successfully matches.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e., zero). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond the end of the string (in which case this routine backtracks to previous functions) or the following match routine(s) succeed.

```
procedure pat.oneOrMoreCset( cst:cset );
```

```
pat.match
<< pattern matching statements >>
pat.oneOrMoreCset( {'0'..'9'} );
<< pattern matching statements >>
pat.endMatch;
```

Matches one or more characters in the specified character set. Immediately fails if there isn't at least one character in *cst*. It advances the cursor beyond all the characters that it successfully matches.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to the original cursor position plus one (in which case this routine fails) or the following match routine(s) succeed.

```
procedure pat.l_OneOrMoreCset( cst:cset );
```

```
pat.match
<< pattern matching statements >>
pat.l_OneOrMoreCset( {'0'..'9'} );
<< pattern matching statements >>
```

```
pat.endMatch;
```

Matches one or more characters in the specified character set. Immediately fails if there isn't at least one character in *cst*. It advances the cursor beyond all the characters that it successfully matches.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e., one). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond the end of the string (in which case this routine fails) or the following match routine(s) succeed.

```
procedure pat.exactlyNCset( cst:cset; n:uns32 );
```

```
pat.match
  << pattern matching statements >>
  pat.exactlyNCset( {'0'..'9'}, 4 );
  << pattern matching statements >>
pat.endMatch;
```

Matches exactly *n* characters that are members of *cst*. If any of the next *n* characters in the match string are not in *cst*, this routine returns failure. This function advances the cursor by *n* positions if it succeeds.

**Note:** The character at position (*n*+1) must *not* be a member of *cst* or this routine fails.

```
procedure pat.firstNCset( cst:cset; n:uns32 );
```

```
pat.match
  << pattern matching statements >>
  pat.firstNCset( {'0'..'9'}, 4 ); // Matches 4 numeric chars in string
  << pattern matching statements >>
pat.endMatch;
```

Matches *n* characters that are members of *cst*. On success this function advances the cursor by *n* positions.

**Note:** The character at position (*n*+1) may be a member of *cst*. Whether or not it is, this routine succeeds if the first *n* characters are members of *cst*.

```
procedure pat.norLessCset( cst:cset; n:uns32 );
```

```
pat.match
  << pattern matching statements >>
  pat.norLessCset( {'0'..'9'}, i ); // Matches 0..i numeric chars
  << pattern matching statements >>
pat.endMatch;
```

This routine matches *n* or fewer characters belonging to the *cst* set. This function always succeeds as it can match zero character (which are less than *n* characters).

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string (up to *n*). If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to the original cursor position (in which case this routine backtracks through any previous pattern matching functions) or the following match routine(s) succeed.

```
procedure pat.l_NorLessCset( cst:cset; n:uns32 );
```

```
pat.match
  << pattern matching statements >>
  pat.l_NorLessCset( {'0'..'9'}, i ); // Matches 0..i numeric chars
  << pattern matching statements >>
pat.endMatch;
```

This routine matches *n* or fewer characters belonging to the *cst* set. This function always succeeds as it can match zero character (which are less than *n* characters).

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e., zero). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues

until it advances beyond the end of the string (in which case this routine backtracks through any previous pattern matching functions) or the following match routine(s) succeed.

```
procedure pat.norMoreCset( cst:cset; n:uns32 );
  pat.match
  << pattern matching statements >>
  pat.norMoreCset( { '0'..'9' }, i ); // Matches i..? numeric chars
  << pattern matching statements >>
  pat.endMatch;
```

This routine matches at least  $n$  characters belonging to the *cst* set. If fewer than  $n$  characters match the set, this routine returns failure. If this function succeeds, it advances the cursor beyond all the characters it matches in *cst*.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to the original cursor position (in which case this routine backtracks through any previous pattern matching functions) or the following match routine(s) succeed.

```
procedure pat.l_NorMoreCset( cst:cset; n:uns32 );
  pat.match
  << pattern matching statements >>
  pat.l_NorMoreCset( { '0'..'9' }, i ); // Matches i..? numeric chars
  << pattern matching statements >>
  pat.endMatch;
```

This routine matches at least  $n$  characters belonging to the *cst* set. If fewer than  $n$  characters match the set, this routine returns failure. If this function succeeds, it advances the cursor beyond all the characters it matches in *cst*.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e.,  $n$ ). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond the end of the string (in which case this routine backtracks through any previous pattern matching functions) or the following match routine(s) succeed.

```
procedure pat.ntoMCset( cst:cset; n:uns32; m:uns32 );
  pat.match
  << pattern matching statements >>
  pat.ntoMCset( { '0'..'9' }, i, j ); // Matches i..j numeric chars
  << pattern matching statements >>
  pat.endMatch;
```

This routine matches at least  $n$  characters and no more than  $m$  characters belonging to the *cst* set. If fewer than  $n$  characters match the set, this routine returns failure. This routine does not fail if more than  $m$  characters belong to the set. However, it only matches through position  $m$ .

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to the original cursor position (in which case this routine backtracks through any previous pattern matching functions) or the following match routine(s) succeed.

```
procedure pat.l_NtoMCset( cst:cset; n:uns32; m:uns32 );
  pat.match
  << pattern matching statements >>
  pat.l_NtoMCset( { '0'..'9' }, i, j ); // Matches i..j numeric chars
  << pattern matching statements >>
  pat.endMatch;
```

This routine matches at least  $n$  characters and no more than  $m$  characters belonging to the *cst* set. If fewer than  $n$  characters match the set, this routine returns failure. This routine does not fail if more than  $m$  characters belong to the set. However, it only matches through position  $m$ .

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e.,  $n$ ). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond position  $m$  (in which case this routine backtracks through any previous pattern matching functions) or the following match routine(s) succeed.

```
procedure pat.exactlyNtoMCset( cst:cset; n:uns32; m:uns32 );

  pat.match
    << pattern matching statements >>
    pat.exactlyNtoMCset( { '0'..'9' }, i, j ); // Matches i..j numeric chars
    << pattern matching statements >>
  pat.endMatch;
```

This routine matches at least  $n$  characters and no more than  $m$  characters belonging to the *cst* set. If fewer than  $n$  characters match the set, this routine returns failure. This routine fails if more than  $m$  characters belong to the set.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to the original cursor position (in which case this routine backtracks through any previous pattern matching functions) or the following match routine(s) succeed.

```
procedure pat.l_ExactlyNtoMCset( cst:cset; n:uns32; m:uns32 );

  pat.match
    << pattern matching statements >>
    pat.l_ExactlyNtoMCset( { '0'..'9' }, i, j );           // Matches i..j
                                                           // numeric chars
    << pattern matching statements >>
  pat.endMatch;
```

This routine matches at least  $n$  characters and no more than  $m$  characters belonging to the *cst* set. If fewer than  $n$  characters match the set, this routine returns failure. This routine fails if more than  $m$  characters belong to the set.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e.,  $n$ ). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond position  $m$  (in which case this routine backtracks through any previous pattern matching functions) or the following match routine(s) succeed.

## 25.15 Character Matching Functions

The following sections describe each of the character matching functions provided by the HLA patterns module. These functions take (at the minimum) a character object. The characters in the match string (at the cursor position) are tested against the character. These functions succeed if, as appropriate for the specific function, they match the parameterized character.

You'll notice that there are no "not equal to character" type functions in this group. You can easily test to see if a string does not match any character using the character set pattern matching functions.

```
procedure pat.peekChar( c:char );

  pat.match
    << pattern matching statements >>
    pat.peekChar( 'a' ); // Matches 'a'
    << pattern matching statements >>
  pat.endMatch;
```

This routine succeeds if the character pointed at by the cursor (ESI) is equal to *c*; it fails otherwise. This routine does not advance the cursor if it succeeds.

```
procedure pat.oneChar( c:char );
    pat.match
    << pattern matching statements >>
    pat.oneChar( 'a' ); // Matches 'a'
    << pattern matching statements >>
    pat.endMatch;
```

This routine succeeds if the character pointed at by the cursor (ESI) is equal to *c*; it fails otherwise. If it succeeds, this routine advances the cursor over the character it matches.

```
procedure pat.upToChar( c:char );
    pat.match
    << pattern matching statements >>
    pat.oneChar( ch ); // Matches char in ch register
    << pattern matching statements >>
    pat.endMatch;
```

This routine matches all characters in a string from the cursor position up to the specified parameter. It fails if the specified character is not in the string. Note that this routine leaves the cursor pointing at the character specified by the parameter (i.e., it still remains to be matched). A call to `pat.extract` immediately after this function will create a string with all the characters up to, but not including, the character passed as the parameter.

```
procedure pat.zeroOrOneChar( c:char );
    pat.match
    << pattern matching statements >>
    pat.zeroOrOneChar( c ); // Matches char in c variable
    << pattern matching statements >>
    pat.endMatch;
```

This routine matches zero or one occurrences of the character parameter. Because it can match zero characters, this function always succeeds. This function is great for matching an optional character in a pattern.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match one character in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine.

```
procedure pat.l_ZeroOrOneChar( c:char );
    pat.match
    << pattern matching statements >>
    pat.zeroOrOneChar( c ); // Matches char in c variable
    << pattern matching statements >>
    pat.endMatch;
```

This routine matches zero or one occurrences of the character parameter. In other words, it lets you check for the presence of an optional character. Because this function can match zero characters, it always succeeds.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e., zero). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. If that fails, then this routine fails.

```
procedure pat.zeroOrMoreChar( c:char );
    pat.match
    << pattern matching statements >>
    pat.zeroOrMoreChar( c ); // Matches char in c variable
    << pattern matching statements >>
    pat.endMatch;
```



This routine matches zero or more occurrences of the character parameter. It leaves the cursor pointing at the end of the string or the first character that is not equal to *c*. Because this function can match zero characters, it always succeeds.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to the original cursor position (in which case this routine fails) or the following match routine(s) succeed.

```
procedure pat.l_ZeroOrMoreChar( c:char );
    pat.match
    << pattern matching statements >>
    pat.l_ZeroOrMoreChar( c ); // Matches char in c variable
    << pattern matching statements >>
    pat.endMatch;
```

This routine matches zero or more occurrences of the character parameter. It leaves the cursor pointing at the end of the string or the first character that is not equal to *c*. Because this function can match zero characters, it always succeeds.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e., zero). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond the end of the string (in which case this routine fails) or the following match routine(s) succeed.

```
procedure pat.oneOrMoreChar( c:char );
    pat.match
    << pattern matching statements >>
    pat.oneOrMoreChar( c ); // Matches char in c variable
    << pattern matching statements >>
    pat.endMatch;
```

This routine matches one or more occurrences of the character parameter. It leaves the cursor pointing at the end of the string or the first character that is not equal to *c*. It fails if there isn't at least one copy of *c* at the cursor position.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to the cursor position of the first character it matched (in which case this routine fails) or the following match routine(s) succeed.

```
procedure pat.l_OneOrMoreChar( c:char );
    pat.match
    << pattern matching statements >>
    pat.l_OneOrMoreChar( c ); // Matches char in c variable
    << pattern matching statements >>
    pat.endMatch;
```

This routine matches one or more occurrences of the character parameter. It leaves the cursor pointing at the end of the string or the first character that is not equal to *c*. It fails if there isn't at least one copy of *c* at the cursor position.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e., one). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond the end of the string (in which case following functions fail and this function returns the failure on back up the invocation chain) or the following match routine(s) succeed.

```
procedure pat.exactlyNChar( c:char; n:uns32 );
    pat.match
    << pattern matching statements >>
    pat.exactlyNChar( c, n ); // Matches char in c variable
    << pattern matching statements >>
```

```
pat.endMatch;
```

This routine matches exactly  $n$  copies of the character  $c$  in the string. If more, or less, copies of  $c$  appear in the string at the current cursor position then this routine fails. Note that the character at cursor position  $(n+1)$  must not be equal to  $c$  or this function fails even if the first  $n$  characters do match.

```
procedure pat.firstNChar( c:char; n:uns32 );
```

```
pat.match
  << pattern matching statements >>
  pat.firstNChar( c, n ); // Matches char in c variable
  << pattern matching statements >>
pat.endMatch;
```

This routine matches  $n$  copies of the character  $c$  in the string. If fewer than  $n$  copies of  $c$  appear in the string, this routine fails. If more copies of  $c$  appear in the string, this routine succeeds, however, it only matches the first  $n$  copies.

```
procedure pat.norLessChar( c:char; n:uns32 );
```

```
pat.match
  << pattern matching statements >>
  pat.norLessChar( c, n ); // Matches char in c variable
  << pattern matching statements >>
pat.endMatch;
```

This procedure matches  $n$  or fewer copies of  $c$  in the current string. If additional copies of  $c$  appear in the string, this routine still succeeds but it only matches the first  $n$  copies.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to the original cursor position (in which case this routine fails) or the following match routine(s) succeed.

```
procedure pat.l_NorLessChar( c:char; n:uns32 );
```

```
pat.match
  << pattern matching statements >>
  pat.l_NorLessChar( c, n ); // Matches char in c variable
  << pattern matching statements >>
pat.endMatch;
```

This procedure matches  $n$  or fewer copies of  $c$  in the current string. If additional copies of  $c$  appear in the string, this routine still succeeds but it only matches the first  $n$  copies.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e., zero). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond the end of the string (in which case this routine passes back the failure returned by the following matching functions) or the following match routine(s) succeed.

```
procedure pat.norMoreChar( c:char; n:uns32 );
```

```
pat.match
  << pattern matching statements >>
  pat.norMoreChar( c, n ); // Matches char in c variable
  << pattern matching statements >>
pat.endMatch;
```

This procedure matches  $n$  or more copies of  $c$  in the current string. It fails if there are fewer than  $n$  copies of  $c$ .

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to position  $n$  (in which case this routine fails) or the following match routine(s) succeed.

```
procedure pat.l_NorMoreChar( c:char; n:uns32 );
```

```
  pat.match
    << pattern matching statements >>
    pat.l_NorMoreChar( c, n ); // Matches char in c variable
    << pattern matching statements >>
  pat.endMatch;
```

This procedure matches  $n$  or more copies of  $c$  in the current string. It fails if there are fewer than  $n$  copies of character  $c$  in the string.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e.,  $n$ ). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond the end of the string (in which case this routine returns the failure reported by the following match routines) or the following match routine(s) succeed.

```
procedure pat.ntoMChar( c:char; n:uns32; m:uns32 );
```

```
  pat.match
    << pattern matching statements >>
    pat.ntoMChar( c, n, m );
    << pattern matching statements >>
  pat.endMatch;
```

This procedure matches between  $n$  and  $m$  copies of the character  $c$  starting at the current cursor (ESI) position. This routine succeeds even if there are more than  $m$  copies of the character, however, it will only match the first  $m$  characters in the string.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to position  $n$  (in which case this routine fails) or the following match routine(s) succeed.

```
procedure pat.l_NtoMChar( c:char; n:uns32; m:uns32 );
```

```
  pat.match
    << pattern matching statements >>
    pat.l_NtoMChar( c, n, m );
    << pattern matching statements >>
  pat.endMatch;
```

This procedure matches between  $n$  and  $m$  copies of the character  $c$  starting at the current cursor (ESI) position. This routine succeeds even if there are more than  $m$  copies of the character, however, it will only match the first  $m$  characters in the string.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e.,  $n$ ). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond position  $m$  (in which case this routine returns the failure) or the following match routine(s) succeed.

```
procedure pat.exactlyNtoMChar( c:char; n:uns32; m:uns32 );
```

```
  pat.match
    << pattern matching statements >>
    pat.exactlyNtoMChar( c, n, m );
    << pattern matching statements >>
  pat.endMatch;
```

This procedure matches between  $n$  and  $m$  copies of the character  $c$  starting at the current cursor (ESI) position. This routine fails if there are more than  $m$  copies (or fewer than  $n$  copies) of the character in the string.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to position  $n$  (in which case this routine fails) or the following match routine(s) succeed.

```

procedure pat.l_ExactlyNtoMChar( c:char; n:uns32; m:uns32 );
    pat.match
    << pattern matching statements >>
    pat.l_ExactlyNtoMChar( c, n, m );
    << pattern matching statements >>
    pat.endMatch;

```

This procedure matches between  $n$  and  $m$  copies of the character  $c$  starting at the current cursor (ESI) position. This routine fails if there are more than  $m$  copies (or fewer than  $n$  copies) of the character in the string.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e.,  $n$ ). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond position  $m$  (in which case this routine fails) or the following match routine(s) succeed.

## 25.16 Case Insensitive Character Matching Routines

These routines are semantically identical to the above routines with one difference- when they compare the characters they use a case insensitive comparison. Please see the descriptions above for an explanation of these routines.

```

procedure pat.peekiChar( c:char );
    pat.match
    << pattern matching statements >>
    pat.peekChar( 'a' ); // Matches 'a'
    << pattern matching statements >>
    pat.endMatch;

```

This routine succeeds if the character pointed at by the cursor (ESI) is equal to  $c$  using a case insensitive comparison; it fails otherwise. This routine does not advance the cursor if it succeeds.

```

procedure pat.oneiChar( c:char );
    pat.match
    << pattern matching statements >>
    pat.oneChar( 'a' ); // Matches 'a'
    << pattern matching statements >>
    pat.endMatch;

```

This routine succeeds if the character pointed at by the cursor (ESI) is equal to  $c$  using a case insensitive comparison; it fails otherwise. If it succeeds, this routine advances the cursor over the character it matches.

```

procedure pat.upToiChar( c:char );
    pat.match
    << pattern matching statements >>
    pat.oneChar( ch ); // Matches char in ch register
    << pattern matching statements >>
    pat.endMatch;

```

This routine matches all characters, using a case insensitive comparison, in a string from the cursor position up to the specified parameter. It fails if the specified character is not in the string. Note that this routine leaves the cursor pointing at the character specified by the parameter (i.e., it still remains to be matched). A call to `pat.extract` immediately after this function will create a string with all the characters up to, but not including, the character passed as the parameter.

```

procedure pat.zeroOrOneiChar( c:char );
    pat.match
    << pattern matching statements >>
    pat.zeroOrOneChar( c ); // Matches char in c variable
    << pattern matching statements >>

```

```
pat.endMatch;
```

This routine matches zero or one occurrences of the character parameter using a case insensitive comparison. Because it can match zero characters, this function always succeeds. This function is great for matching an optional character in a pattern.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match one character in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine.

```
procedure pat.l_ZeroOrOneiChar( c:char );
    pat.match
    << pattern matching statements >>
    pat.zeroOrOneChar( c ); // Matches char in c variable
    << pattern matching statements >>
    pat.endMatch;
```

This routine matches zero or one occurrences of the character parameter using a case insensitive comparison. In other words, it lets you check for the presence of an optional character. Because this function can match zero characters, it always succeeds.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e., zero). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. If that fails, then this routine fails.

```
procedure pat.zeroOrMoreiChar( c:char );
    pat.match
    << pattern matching statements >>
    pat.zeroOrMoreChar( c ); // Matches char in c variable
    << pattern matching statements >>
    pat.endMatch;
```

This routine matches zero or more occurrences of the character parameter using a case insensitive comparison. It leaves the cursor pointing at the end of the string or the first character that is not equal to *c*. Because this function can match zero characters, it always succeeds.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to the original cursor position (in which case this routine fails) or the following match routine(s) succeed.

```
procedure pat.l_ZeroOrMoreiChar( c:char );
    pat.match
    << pattern matching statements >>
    pat.l_ZeroOrMoreChar( c ); // Matches char in c variable
    << pattern matching statements >>
    pat.endMatch;
```

This routine matches zero or more occurrences of the character parameter using a case insensitive comparison. It leaves the cursor pointing at the end of the string or the first character that is not equal to *c*. Because this function can match zero characters, it always succeeds.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e., zero). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond the end of the string (in which case this routine fails) or the following match routine(s) succeed.

```
procedure pat.oneOrMoreiChar( c:char );
    pat.match
    << pattern matching statements >>
    pat.oneOrMoreChar( c ); // Matches char in c variable
    << pattern matching statements >>
```

```
pat.endMatch;
```

This routine matches one or more occurrences of the character parameter using a case insensitive comparison. It leaves the cursor pointing at the end of the string or the first character that is not equal to *c*. It fails if there isn't at least one copy of *c* at the cursor position.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to the cursor position of the first character it matched (in which case this routine fails) or the following match routine(s) succeed.

```
procedure pat.l_OneOrMoreiChar( c:char );
    pat.match
    << pattern matching statements >>
    pat.l_OneOrMoreChar( c ); // Matches char in c variable
    << pattern matching statements >>
    pat.endMatch;
```

This routine matches one or more occurrences of the character parameter using a case insensitive comparison. It leaves the cursor pointing at the end of the string or the first character that is not equal to *c*. It fails if there isn't at least one copy of *c* at the cursor position.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e., one). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond the end of the string (in which case following functions fail and this function returns the failure on back up the invocation chain) or the following match routine(s) succeed.

```
procedure pat.exactlyNiChar( c:char; n:uns32 );
    pat.match
    << pattern matching statements >>
    pat.exactlyNChar( c, n ); // Matches char in c variable
    << pattern matching statements >>
    pat.endMatch;
```

This routine matches exactly *n* copies of the character *c* in the string using a case insensitive comparison. If more, or less, copies of *c* appear in the string at the current cursor position then this routine fails. Note that the character at cursor position (*n*+1) must not be equal to *c* or this function fails even if the first *n* characters do match.

```
procedure pat.firstNiChar( c:char; n:uns32 );
    pat.match
    << pattern matching statements >>
    pat.firstNChar( c, n ); // Matches char in c variable
    << pattern matching statements >>
    pat.endMatch;
```

This routine matches *n* copies of the character *c* in the string using a case insensitive comparison. If fewer than *n* copies of *c* appear in the string, this routine fails. If more copies of *c* appear in the string, this routine succeeds, however, it only matches the first *n* copies.

```
procedure pat.norLessiChar( c:char; n:uns32 );
    pat.match
    << pattern matching statements >>
    pat.norLessChar( c, n ); // Matches char in c variable
    << pattern matching statements >>
    pat.endMatch;
```

This procedure matches *n* or fewer copies of *c* in the current string using a case insensitive comparison. If additional copies of *c* appear in the string, this routine still succeeds but it only matches the first *n* copies.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to the original cursor position (in which case this routine fails) or the following match routine(s) succeed.

```
procedure pat.l_NorLessiChar( c:char; n:uns32 );
```

```
  pat.match
    << pattern matching statements >>
    pat.l_NorLessChar( c, n ); // Matches char in c variable
    << pattern matching statements >>
  pat.endMatch;
```

This procedure matches  $n$  or fewer copies of  $c$  in the current string using a case insensitive comparison. If additional copies of  $c$  appear in the string, this routine still succeeds but it only matches the first  $n$  copies.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e., zero). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond the end of the string (in which case this routine passes back the failure returned by the following matching functions) or the following match routine(s) succeed.

```
procedure pat.norMoreiChar( c:char; n:uns32 );
```

```
  pat.match
    << pattern matching statements >>
    pat.norMoreChar( c, n ); // Matches char in c variable
    << pattern matching statements >>
  pat.endMatch;
```

This procedure matches  $n$  or more copies of  $c$  in the current string using a case insensitive comparison. It fails if there are fewer than  $n$  copies of  $c$ .

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to position  $n$  (in which case this routine fails) or the following match routine(s) succeed.

```
procedure pat.l_NorMoreiChar( c:char; n:uns32 );
```

```
  pat.match
    << pattern matching statements >>
    pat.l_NorMoreChar( c, n ); // Matches char in c variable
    << pattern matching statements >>
  pat.endMatch;
```

This procedure matches  $n$  or more copies of  $c$  in the current string using a case insensitive comparison. It fails if there are fewer than  $n$  copies of character  $c$  in the string.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e.,  $n$ ). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond the end of the string (in which case this routine returns the failure reported by the following match routines) or the following match routine(s) succeed.

```
procedure pat.ntoMiChar( c:char; n:uns32; m:uns32 );
```

```
  pat.match
    << pattern matching statements >>
    pat.ntoMChar( c, n, m );
    << pattern matching statements >>
  pat.endMatch;
```

This procedure matches between  $n$  and  $m$  copies of the character  $c$  starting at the current cursor (ESI) position, using a case insensitive comparison. This routine succeeds even if there are more than  $m$  copies of the character, however, it will only match the first  $m$  characters in the string.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to position  $n$  (in which case this routine fails) or the following match routine(s) succeed.

```
procedure pat.l_NtoMiChar( c:char; n:uns32; m:uns32 );
```

```
  pat.match
    << pattern matching statements >>
    pat.l_NtoMChar( c, n, m );
    << pattern matching statements >>
  pat.endMatch;
```

This procedure matches between  $n$  and  $m$  copies of the character  $c$  starting at the current cursor (ESI) position, using a case insensitive comparison. This routine succeeds even if there are more than  $m$  copies of the character, however, it will only match the first  $m$  characters in the string.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e.,  $n$ ). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond position  $m$  (in which case this routine returns the failure) or the following match routine(s) succeed.

```
procedure pat.exactlyNtoMiChar( c:char; n:uns32; m:uns32 );
```

```
  pat.match
    << pattern matching statements >>
    pat.exactlyNtoMChar( c, n, m );
    << pattern matching statements >>
  pat.endMatch;
```

This procedure matches between  $n$  and  $m$  copies of the character  $c$  starting at the current cursor (ESI) position, using a case insensitive comparison. This routine fails if there are more than  $m$  copies (or fewer than  $n$  copies) of the character in the string.

This function uses an "aggressive" or "eager" pattern matching algorithm. It will attempt to match as many characters as possible in the string. If doing so would cause a following match routine to fail, this routine will backtrack one character and retry the following match routine. This continues until it backs up to position  $n$  (in which case this routine fails) or the following match routine(s) succeed.

```
procedure pat.l_ExactlyNtoMiChar( c:char; n:uns32; m:uns32 );
```

```
  pat.match
    << pattern matching statements >>
    pat.l_ExactlyNtoMChar( c, n, m );
    << pattern matching statements >>
  pat.endMatch;
```

This procedure matches between  $n$  and  $m$  copies of the character  $c$  starting at the current cursor (ESI) position, using a case insensitive comparison. This routine fails if there are more than  $m$  copies (or fewer than  $n$  copies) of the character in the string.

This function uses a "deferred" or "lazy" pattern matching algorithm. It will attempt to match as few characters as possible in the string (i.e.,  $n$ ). If doing so would cause a following match routine to fail, this routine will backtrack and advance one character and then retry the following match routine. This continues until it advances beyond position  $m$  (in which case this routine fails) or the following match routine(s) succeed.

#### String Matching Functions

```
procedure pat.matchStr( s:string );
```

```
  pat.match
    << pattern matching statements >>
    pat.matchStr( someString );
    << pattern matching statements >>
  pat.endMatch;
```



If the sequence of characters at the current cursor position (ESI) match the specified string, this routine succeeds, otherwise it fails. Note that additional characters may appear in the match string after the characters matched by *s*.

```
procedure pat.matchiStr( s:string );
    pat.match
    << pattern matching statements >>
    pat.matchiStr( someString );
    << pattern matching statements >>
    pat.endMatch;
```

Like *pat.matchStr*, except this routine does a case insensitive comparison.

```
procedure pat.matchToStr( s:string );
    pat.match
    << pattern matching statements >>
    pat.matchToStr( someString );
    << pattern matching statements >>
    pat.endMatch;
```

This routine matches all characters up to, and including, the parameter string *s*. If it matches a string and a following pattern matching routine fails, this routine handles the backtracking and searches for the next string that matches. The backtracking is lazy insofar as this routine will always match the minimum number of characters up to *s* in the string in order to succeed. When backtracking occurs, this function will skip over the string it has matched and search for another occurrence. This function will fail if it cannot find another occurrence of *s* in the match string.

```
procedure pat.upToStr( s:string );
    pat.match
    << pattern matching statements >>
    pat.upToStr( someString );
    << pattern matching statements >>
    pat.endMatch;
```

This routine matches all characters up to, but not including, the parameter string *s*. If it matches a string and a following pattern matching routine fails, this routine handles the backtracking and searches for the next string that matches. The backtracking is lazy insofar as this routine will always match the minimum number of occurrences of *s* in the string in order to succeed.

```
procedure pat.matchToiStr( s:string );
    pat.match
    << pattern matching statements >>
    pat.matchToiStr( someString );
    << pattern matching statements >>
    pat.endMatch;
```

Like *pat.matchToStr*, except this routine does a case insensitive comparison.

```
procedure pat.upToiStr( s:string );
    pat.match
    << pattern matching statements >>
    pat.upToiStr( someString );
    << pattern matching statements >>
    pat.endMatch;
```

Like *pat.upToStr*, except this routine does a case insensitive comparison.

```
procedure pat.matchWord( s:string );
    pat.match
    << pattern matching statements >>
```

```

pat.matchWord( someString );
<< pattern matching statements >>
pat.endMatch;

```

This routine is similar to *pat.matchStr* except that it requires a delimiter character after the string it matches. The delimiter character is a member of the *WordDelims* character set (internal to the patterns.hhf code). *WordDelims* is, by default, the character set "-{'a'..'z', 'A'..'Z', '0'..'9', '\_'}" (that is, all character except the alphanumeric characters and the underscore). See the *getWordDelims* and *setWordDelims* procedures if you are interested in changing the word delimiters set.

```

procedure pat.matchiWord( s:string );
  pat.match
  << pattern matching statements >>
  pat.matchiWord( someString );
  << pattern matching statements >>
  pat.endMatch;

```

Just like *pat.matchWord*, except this routine does a case insensitive comparison.

```

procedure pat.getWordDelims( var cst:cset );
  pat.getWordDelims( destinationCSet );

```

This function makes a copy of the internal *WordDelims* character set and places this copy in the specified *cst* parameter. Note that you do not have to call this function inside a *pat.match..pat.endMatch* sequence (though it's perfectly okay to do so).

```

procedure pat.setWordDelims( cst:cset);
  pat.setWordDelims( newDelimsCSet );

```

This function stores the value of the *cst* character set into the *WordDelims* character set. This allows you to change the *WordDelims* character set to your liking. Note that you do not have to call this function inside a *pat.match..pat.endMatch* sequence (though it's perfectly okay to do so).

## 25.17 String Extraction Functions

```

procedure pat.extract( s:string );
  pat.match
  << pattern matching statements >>
  pat.extract( someAllocatedStringObject );
  << pattern matching statements >>
  pat.endMatch;

```

Whenever a pattern matching routine successfully matches zero or more characters in the string, the pattern matching routine returns a pointer to the start of the matched characters in EBX and a pointer to the position just beyond the last matched position in ESI. You may use the *pat.extract* procedure to create an HLA-compatible string of these matched characters. This routine will raise an exception if the destination string isn't big enough to hold the extracted characters.

Note that *pat.extract* will only extract those characters that the immediately previous string matching function matched. If you want to extract a string from a sequence of match functions, use the *pat.onePat..pat.endOnePat* sequence to group the functions whose matched string you want to extract.

Be careful about making calls to *pat.extract* when backtracking can occur. Though *pat.extract* will work fine in the event of backtracking, you will take a big performance hit if the system has to make a copy of the same string over and over again if backtracking occurs frequently.

**Warning:** *pat.extract* should only be called in the "success" section of a *pat.match..pat.endmatch* block. Any other invocation could create a problem. In general, you must ensure that EBX and ESI point at reasonable spots within the same string. Note that pattern match failure does not guarantee that EBX contains a reasonable value. Therefore, you should not use *pat.extract* at a point where string failure could have occurred unless you explicitly set up EBX (and, possibly, ESI) yourself.

```

procedure pat.a_extract( var s:string );
    pat.match
    << pattern matching statements >>
    pat.a_extract();
    mov( eax, stringVariable );
    << pattern matching statements >>
    pat.endMatch;

```

Whenever a pattern matching routine successfully matches zero or more characters in the string, the pattern matching routine returns a pointer to the start of the matched characters in EBX and a pointer to the position just beyond the last matched position in ESI. You may use the *pat.a\_extract* procedure to create an HLA-compatible string of these matched characters. *pat.a\_extract* will allocate storage for the string on the heap, copy the matched characters to this string, and then store a pointer to the new string in the string variable passed as a reference parameter to *pat.a\_extract*.

Be careful about making calls to *pat.a\_extract* when backtracking can occur. Though *pat.a\_extract* will work fine in the event of backtracking, you will take a big performance hit if the system has to make a copy of the same string over and over again if backtracking occurs frequently. Also, note that unless you take care to free the string data allocated on a previous call to *pat.a\_extract* (before the backtracking occurs), you'll wind up with a "memory leak". For this reason, you should use *pat.extract* on a preallocated string rather than calling *pat.a\_extract* to allocate the string.

**Warning:** *pat.a\_extract* should only be called in the "success" section of a *pat.match..pat.endmatch* block. Any other invocation could create a problem. In general, you must ensure that EBX and ESI point at reasonable spots within the same string. Note that pattern match failure does not guarantee that EBX contains a reasonable value. Therefore, you should not use *pat.a\_extract* at a point where string failure could have occurred unless you explicitly set up EBX (and, possibly, ESI) yourself.

## 25.18 Whitespace and End of String Matching Functions

These convenient routines match a sequence of whitespace characters as well as the end of the current string. By default, these routines assume that whitespace consists of all the control characters, the ASCII space (#\$20), and the del code (#\$7f). You can change this definition using the *pat.getWhiteSpace* and *pat.setWhiteSpace* procedures.

```

procedure pat.getWhiteSpace( var cst:cset );
    pat.getWhiteSpace( destinationCSet );

```

This function returns the current value of the internal *WhiteSpace* character set. It stores the result in the reference parameter *cst*. Note that you do not have to call this function inside a *pat.match..pat.endMatch* sequence (though it's perfectly okay to do so).

```

procedure pat.setWhiteSpace( cst:cset);
    pat.setWhiteSpace( newCSet );

```

This procedure copies the specified character set to the internal *WhiteSpace* character set. All future whitespace matching procedures will use this new value when matching white space characters. Note that you do not have to call this function inside a *pat.match..pat.endMatch* sequence (though it's perfectly okay to do so).

```

procedure pat.zeroOrMoreWS;
    pat.match
    << pattern matching statements >>
    pat.zeroOrMoreWS();
    << pattern matching statements >>
    pat.endMatch;

```

This routine matches zero or more whitespace characters. This routine uses an "eager" matching algorithm.

**procedure pat.oneOrMoreWS;**

```

pat.match
  << pattern matching statements >>
  pat.oneOrMoreWS();
  << pattern matching statements >>
pat.endMatch;

```

This routine matches zero or more whitespace characters. This routine uses an "eager" matching algorithm; it will backtrack over matched white space characters at the end if the following match functions require some whitespace characters to succeed. If there isn't at least one whitespace character at the cursor position, this function fails. Otherwise, it succeeds.

**procedure pat.WSorEOS;**

```

pat.match
  << pattern matching statements >>
  pat.WSorEOS();
  << pattern matching statements >>
pat.endMatch;

```

This routine matches a single whitespace character or the end of the string. It fails if there are characters left in the string and the character at the cursor position is not a white space character.

**procedure pat.WSthenEOS;**

```

pat.match
  << pattern matching statements >>
  pat.WSthenWS();
  << pattern matching statements >>
pat.endMatch;

```

This routine matches zero or more white space characters that appear at the end of the current string. It fails if there are any other characters before the end of the string.

**procedure pat.peekWS;**

```

pat.match
  << pattern matching statements >>
  pat.peekWS();
  << pattern matching statements >>
pat.endMatch;

```

This routine succeeds if the next character in the string is a whitespace character. However, it does not advance the cursor over the character.

**procedure pat.peekWSorEOS;**

```

pat.match
  << pattern matching statements >>
  pat.peekWSorEOS();
  << pattern matching statements >>
pat.endMatch;

```

This routine succeeds if the next character in the string is a white space character or if there are no more characters in the string. It does not advance the cursor.

## 25.19 Matching an Arbitrary Sequence of Characters

**procedure pat.arb;**

```

pat.match
  << pattern matching statements >>

```

```

pat.arb();
<< pattern matching statements >>
pat.endMatch;

```

This routine matches zero or more characters. It uses an "aggressive" or "eager" matching algorithm, immediately matching all the remaining characters in the string. If following matching routines fail, this routine backtracks one character at a time until reaching the initial starting position (in which case this routine fails) or the following matching routine(s) succeed.

**procedure pat.l\_arb; external;**

```

pat.match
  << pattern matching statements >>
  pat.l_arb();
  << pattern matching statements >>
pat.endMatch;

```

This is a "lazy" or "deferred" version of the above routine. It matches zero characters and succeeds; if a following match routine fails, this routine backtracks by advancing the cursor one position for each failure. If this routine advances beyond the end of the string during backtracking, it reports failure.

## 25.20 Writing Your Own Pattern Matching Routines

Although HLA provides a wide variety of pattern matching functions, from which you can probably synthesize any pattern you desire, there are several reasons why you might want to write your own pattern matching routines. Some common reasons include: (1) You would like a more efficient pattern matching function than is possible by composing existing pattern matching functions. (2) You need a particular pattern matching routine to produce a side effect and the standard matching routines do not produce the desired side effect. A common example is a pattern matching routine that returns an attribute value for an item it matches. For example, a routine that matches a string of decimal digits may return the numeric equivalent of that string as an attribute of that pattern. (3) You need a pattern matching routine that considers other machine states (i.e., variable values) besides the string the pattern is processing. (4) You need to handle some context-sensitive issues. (5) You want to understand how the pattern matching algorithm works. Writing your own pattern matching functions can achieve all these goals and many more.

The first issue you must address when writing your own pattern matching routine is whether or not the routine supports backtracking. Generally, this decision depends upon whether the function matches strings that are always a fixed length or can match strings of differing lengths. For example, the *pat.oneCset* routine always matches a string of length one whereas the *pat.zeroOrMoreCset* function can match strings of any length. If a function can only match strings having a fixed length, then the function does not need to support backtracking. Generally, pattern matching functions that can match strings of varying lengths should support backtracking<sup>4</sup>. Since supporting backtracking is more work and less efficient, you should only support it when necessary.

Once you've decided that you're going to support backtracking in a matching function, the next issue that concerns you is whether the function supports eager evaluation or lazy/deferred evaluation. (Note: when writing general matching routines for library use, it's generally a good idea to supply two functions, one that supports eager evaluation and one that supports lazy/deferred evaluation.)

A function that supports eager evaluation tries to match the longest possible string when the program calls the function. If the function succeeds and a later matching function fails (invoking the backtracking operation), then the matching function backs off the maximum number of characters that will still match. This process continues until the following code succeeds or the function backs off so much that it, too, fails.

If function that support lazy/deferred evaluations tries to match the shortest possible string. Once it matches the shortest string it can, it passes control on to the following pattern matching functions. If they fail and back tracking returns

---

4. Although this is your decision. If for some reason you don't want to support backtracking in such functions, that is always an option you can choose.

control to the function, it tries to match the next smallest string larger than the one it currently matches. This process repeats until the following match functions succeed or the current function fails to match anything.

Note that the choice of eager vs. lazy/deferred evaluation does not generally affect whether a pattern will match a given string<sup>5</sup>. It does, however, affect the efficiency of the pattern matching operation. Backtracking is a relatively slow operation. If an eager match causes the following pattern functions to fail until the current pattern matching function backs off to the shortest possible string it can match, the program will run much slower than one that uses lazy evaluation for the function (since it starts with the shortest possible string to begin with). On the other hand, if a function needs to match the longest possible string in order for the following matching functions to succeed, choosing lazy evaluation will run much more slowly than eager evaluation. Therefore, the choice of which form is best to use is completely data dependent. If you have no idea which evaluation form should be better, choose eager evaluation since it is more intuitive to those defining the pattern to match.

All pattern matching routines have two implicit parameters passed to them in the ESI and EDI registers. ESI is the current *cursor* position while EDI points at the byte immediately after the last character available for matching. That is, the characters between locations ESI and (EDI-1) form the string to match against the pattern.

The primary purpose of a pattern matching function is to return "success" or "failure" depending upon whether the pattern matches the characters in the string (or however else you define "success" versus "failure"). In addition to returning success or failure, pattern matching functions must also return certain values in some of the registers. In particular, the function must preserve the value in EDI (that is, it must still point at the first byte beyond the end of the string to match). If the function succeeds, it must return EBX pointing at the start of the sequence it matched (i.e., EBX must contain the original value in ESI) and ESI must point at the first character beyond the string matched by the function (so the string matched is between addresses EBX and ESI-1). If the function fails, it must return the original values of ESI and EDI in these two registers. EBX's value is irrelevant if the function fails. Except for EBP, the routine need not preserve any other register values (and, in fact, a pattern matching function can use the other registers to return attribute values to the calling code)<sup>6</sup>.

Pattern matching routines that do not support backtracking are the easiest to create and understand. Therefore, it makes sense to begin with a discussion of those types of pattern matching routines.

A pattern matching routine that does not support backtracking succeeds by simply returning to its caller (with the registers containing the appropriate values noted above). If the function fails to match the characters between ESI and (EDI-1), it must call the *pat.\_fail\_* function passing the *pat.FailTo* object as its parameter, e.g.,

```
pat._fail_( pat.FailTo );
```

As a concrete example, consider the following implementation of the *pat.matchStr* function:

```
unit patterns;
#include( "pat.hhf" );

procedure pat.matchStr( s:string ); @nodisplay; @noframe;
begin matchStr;
```

---

5. The one exception has to do with fences. If you set a fence after the pattern matching routine, then backtracking cannot return into the pattern matching function. In this one case, the choice of deferred vs. eager evaluation will have an impact on whether the whole pattern will match a given string.

6. The HLA Standard Library Pattern Matching routines preserve EDX, so this is probably a good convention to follow so you don't surprise your users.

```

push( ebp );          // must do this ourselves since noframe
mov( esp, ebp );      // is specified as an option.
cld();

// Move a copy of ESI into EBX since we need to return
// the starting position in EBX if we succeed.

mov( esi, ebx );

// Compute the length of the remaining
// characters in the sequence we are attempting
// to match (i.e., EDI-ESI) and compare this against
// the length of the string passed as a parameter.
// If the parameter string is longer than the number
// of characters left to match, then we can immediately
// fail since there is no way the string is going to
// to match the string parameter.

mov( s, edx );
mov( (type str.strRec [edx]).length, ecx );
mov( edi, eax );
sub( esi, eax );
if( ecx > eax ) then

    // At this point, there aren't enough characters left
    // in the sequence to match s, so fail.

    pat._fail_( pat.FailTo );

endif;

// Okay, compare the two strings up to the length of s
// to see if they match.

push( edi );
mov( edx, edi );
repe.cmpsb();
pop( edi );
if( @ne ) then

    // At this point, the strings are unequal, so fail.
    // Note that this code must restore ESI to its
    // original value if it returns failure.

    mov( ebx, esi );
    pat._fail_( pat.FailTo );

endif;

// Since this routine doesn't have to handle backtracking,
// a simple return indicates success.

pop( ebp );
ret();

end matchStr;
end patterns;

```

If your function needs to support backtracking, the code will be a little more complex. First of all, your function cannot return to its caller by using the RET instruction. To support backtracking, the function must leave its activation record on the stack when it returns. This is necessary so that when backtracking occurs, the

function can pick up where it left off. It is up to the *pat.match* macro to clean up the stack after a sequence of pattern matching functions successfully match a string.

If a pattern matching function supports backtracking, it must preserve the values of ESP, ESI, and EDI upon initial entry into the code. It will also need to maintain the current cursor position during backtracking and it will need to reserve storage for a special *pat.FailRec* data structure. Therefore, almost every pattern matching routine you'll write that supports backtracking will have the following VAR objects:

```
var
  cursor:    misc.pChar; // Save last matched posn here.
  startPosn: misc.pChar; // Save start of str here.
  endStr:    misc.pChar; // End of string goes here.
  espSave:   dword;      // To clean stk after back trk.
  FailToSave:pat.FailRec; // Save global FailTo value here.
```

Warning: you *must* declare these variables in the VAR section; they must not be static objects.

Upon reentry from backtracking, the ESP register will not contain an appropriate value. It is your code's responsibility to clean up the stack when backtracking occurs. The easiest way to do this is to save a copy of ESP upon initial entry into your function (in the *espSave* variable above) and restore ESP from this value whenever backtracking returns control to your function (you'll see how this happens in a moment). Likewise, upon reentry into your function via backtracking, the registers are effectively scrambled. Therefore, you will need to save ESI's value into the *startPosn* variable and EDI's value into the *endStr* variable upon initial entry into the function. The *startPosn* variable contains the value that EBX must have whenever your function returns success. The *cursor* variable contains ESI's value after you've successfully matched some number of characters. This is the value you reload into ESI whenever backtracking occurs. The *FailToSave* data structure holds important pattern matching information. The pattern matching library automatically fills in this structure when you signal success; you are only responsible for supplying this storage, you do not have to initialize it.

You signal failure in a function that supports backtracking the same way you signaled failure in a routine that does not support backtracking: by invoking *pat.\_fail\_ (pat.FailTo )*; Since your code is failing, the caller will clean up the stack (including removing the local variables you've just allocated and initialized). If the pattern matching system calls your pattern matching function after backtracking occurs, it will reenter your function at its standard entry point where you will, once again, allocate storage for the local variables above and initialize them as appropriate.

If your function succeeds, it usually signals success by invoking the *pat.\_success\_* macro. This macro invocation takes the following form:

```
pat._success_( FailToSave, FailToHere );
```

The first parameter is the *pat.FailRec* object you declared as a local variable in your function. The *pat.\_success\_* macro stores away important information into this object before returning control to the caller. The *FailToHere* symbol is a statement label in your function. If backtracking occurs, control transfers to this label in your function (i.e., this is the backtracking reentry point). The code at the *FailToHere* label must immediately reload ESP from *espSave*, EDI from *endStr*, EBX from *startPosn*, and ESI from *cursor*. Then it does whatever is necessary for the backtrack operation and attempts to succeed or fail again.

The *pat.\_success\_* macro (currently) takes the following form<sup>7</sup>:

```
// The following macro is a utility for
// the pattern matching procedures.
// It saves the current global "FailTo"
// value in the "FailRec" variable specified
// as the first parameter and sets up
// FailTo to properly return control into
// the current procedure at the "FailTarget"
// address. Then it jumps indirectly through
// the procedure's return address to transfer
// control to the next (code sequential)
// pattern matching routine.

#macro _success_( _s_FTSave_, _s_FailTarget_ );
```

7. This code was copied out of the "patterns.hhf" file at the time this document was written. You might want to take a look at the patterns.hhf header file to ensure that this code has not changed since this document was written.



```

// Preserve the old FailTo object in the local
// FailTo variable.

mov( pat.FailTo.ebpSave, _s_FTSave_.ebpSave );
mov( pat.FailTo.jumpAdrs, _s_FTSave_.jumpAdrs );

// Save current EBP and failto target address
// in the global FailTo variable so backtracking
// will return the the current routine.

mov( ebp, pat.FailTo.ebpSave );
mov( &_s_FailTarget_, pat.FailTo.jumpAdrs );

// Push the return address onto the stack (so we
// can return to the caller) and restore
// EBP to the caller's value. Then jump
// back to the caller without cleaning up
// the current routine's stack.

push( [ebp+4] );
mov( [ebp], ebp );
ret();

#endmacro

```

As you can see, this code copies the global *pat.FailTo* object into the *FailToSave* data structure you've created. The *FailTo* structure contains the EBP value and the reentry address of the most recent function that supports backtracking. Your code must save these values in the event your code (ultimately) fails and needs to backtrack to some previous pattern matching function.

After preserving the old value of the global *pat.FailTo* variable, the code above copies EBP and the address of the *FailToHere* label you've specified into the global *pat.FailTo* object.

Finally, the code above returns to the user, without cleaning up the stack, by pushing the return address (so it's on the top of the stack) and restoring the caller's EBP value. The RET instruction above returns control to the function's caller (note that the original return address is still on the stack, the pattern matching routines will never use it).

Should backtracking occur and the program reenters your pattern matching function, it will reenter at the address specified by the second parameter of the *pat.\_success\_* macro (as noted above). You should restore the appropriate register (as noted above) and use the value in the *cursor* variable to determine how to proceed with the backtracking operation. When doing eager evaluation, you will generally need to decrement the value obtained from *cursor* to back off on the length of the string your program has matched (failing if you decrement back to the value in *startPosn*). When doing lazy evaluation, you generally need to increment the value obtained from the *cursor* variable in order to match a longer string (failing if you increment *cursor* to the point it becomes equal to *endStr*).

When executing code in the reentry section of your procedure, the failure and success operations are a little different. Prior to failing, you must manually restore the value in *pat.FailTo* that *pat.\_success\_* saved into the *FailToSave* local variable. You must also restore ESI with the original starting position of the string. The following instruction sequence will accomplish this:

```

// Need to restore FailTo address because it
// currently points at us. We want to jump
// to the correct location.

mov( startPosn, esi );
mov( FailToSave.ebpSave, pat.FailTo.ebpSave );
mov( FailToSave.jumpAdrs, pat.FailTo.jumpAdrs );
pat._fail_( pat.FailTo );

```

Likewise, succeeding in the backtrack reentry section of your program is a little different. You do not want to invoke the *pat.\_success\_* macro because it will overwrite the *FailToSave* value with the global *pat.FailTo*. The global value, however, points at your routine; were you to overwrite this value you'd never be able to fail

back to previous matching functions in the current pattern match. Therefore, you should always execute code like the following when succeeding in the reentry section of your code:

```

mov( esi, cursor ); //Save current cursor value.
push( [ebp+4] );    //Make a copy of the rtn adrs.
mov( [ebp], ebp );  //Restore caller's EBP value.
ret();              //Return to caller.

```

The following is the code for the *pat.oneOrMoreCset* routine (that does an eager evaluation) that demonstrates pattern matching with backtracking.

```

unit patterns;
#include( "pat.hhf" );

/*****
/*
/* OneOrMoreCset-
/*
/* Matches one or more characters in a string from
/* the specified character set.
/*
/* Disposition: Eager
/* BackTrackable: Yes
/*
/* Entry Parameters:
/*
/* ESI: Pointer to sequence of characters to match.
/* EDI: Pointer to byte beyond last char to match.
/* cst: Character set to match with.
/*
/* Exit Parameters (if success):
/*
/* EBX: Points at the start of matched sequence.
/* ESI: Points at first character not in cst.
/* EDI: Unchanged from entry value.
/*
/* Exit Parameters (if failure):
/*
/* EDI: Unchanged from entry value.
/*
/* Unless noted, assume all other registers can be modified
/* by this code.
/*
*****/

procedure pat.oneOrMoreCset( cst:cset ); @nodisplay;
var
    cursor:    misc.pChar;    // Save last matched posn here.
    startPosn: misc.pChar;    // Save start of str here.
    endStr:    misc.pChar;    // End of string goes here.
    espSave:   dword;         // To clean stk after back trk.
    FailToSave: pat.FailRec;   // Save global FailTo value here.

begin oneOrMoreCset;

    // If some routine after this one fails and transfers
    // control via backtracking to this code, the stack

```

```

// will be a mess. So save esp so we can clean up
// the stack if backtracking is necessary.

mov( esp, espSave );

// Save the pointer to the start of the string
// to match. This is used as a "fence" value
// to prevent backtracking past the start of
// the string if things go really wrong.

mov( esi, startPosn );
mov( esi, ebx );

// Save pointer to end of string to match.
// This is needed to restore this value when
// backtracking occurs.

mov( edi, endStr );

// Okay, eagerly match as many characters in
// the character set as possible.

xor( eax, eax );
dec( esi );
repeat
    inc( esi );                // Move to next char in string.
    breakif( esi >= edi );    // Stop at end of string.
    mov( [esi], al );         // Get the char to test.
    bt( eax, (type dword cst)); // See if in cst.

until( @nc ); // Carry is set if al in cst.

// So we can easily back track, save a pointer
// to the first non-matching character.

mov( esi, cursor );

// If we matched at least one character, then
// succeed by jumping to the return address, without
// cleaning up the stack (we need to leave our
// activation record laying around in the event
// backtracking is necessary).

if( esi > ebx ) then
    pat._success_( FailToSave, FailToHere );
endif;

// If we get down here, we didn't match at
// least one character. So transfer control
// to the previous routine that supported
// backtracking.

mov( startPosn, esi );
pat._fail_( pat.FailTo );

```

```

// If someone after us fails and invokes
// backtracking, control is transferred to
// this point. First, we need to restore
// ESP to clean up the junk on the stack.
// Then we back up one character, failing
// if we move beyond the beginning of the
// string. If we don't fail, we jump to
// the code following the call to this
// routine (having backtracked one character).

FailToHere:

    mov( espSave, esp );    // Clean up stack.

    mov( cursor, esi );    // Get last posn we matched.
    dec( esi );            // Back up to prev matched char.
    mov( endStr, edi );
    mov( startPosn, ebx );
    if( esi <= ebx ) then

        // We've backed up to the beginning of
        // the string. So we won't be able to
        // match at least one character.

        mov( ebx, esi );
        mov( FailToSave.ebpSave, pat.FailTo.ebpSave );
        mov( FailToSave.jumpAdrs, pat.FailTo.jumpAdrs );
        pat._fail_( pat.FailTo );

    endif;

    // If we drop down here, there is at least one
    // character left in the string that we've
    // matched, so call the next matching routine
    // (by jumping to the return address) to continue
    // the pattern match.

    mov( esi, cursor );
    mov( [ebp+4], eax );
    mov( [ebp], ebp );
    jmp( eax );

end oneOrMoreCset;

end patterns;

```

The following example code demonstrates the *pat.l\_OneOrMoreCset* routine. This is the same routine as the code above except this code supports lazy/deferred evaluation rather than eager evaluation.

```

unit patterns;
#include( "pat.hhf" );

/*****
/*
/* l_OneOrMoreCset-
*/
*/

```

```

/*                                     */
/* Matches one or more characters in a string from          */
/* the specified character set. Matches the shortest       */
/* possible string that yields (overall) success.          */
/*                                     */
/* Disposition:      Deferred                                     */
/* BackTrackable:    Yes                                       */
/*                                     */
/* Entry Parameters:                                     */
/*                                     */
/* ESI:      Pointer to sequence of characters to match.    */
/* EDI:      Pointer to byte beyond last char to match.    */
/* cst:      Character set to match with.                   */
/*                                     */
/* Exit Parameters (if success):                             */
/*                                     */
/* ESI:      Points at first character not in cst.          */
/* EDI:      Unchanged from entry value.                   */
/*                                     */
/* Exit Parameters (if failure):                             */
/*                                     */
/* EDI:      Unchanged from entry value.                   */
/* ESI:      Unchanged from entry value.                   */
/*                                     */
/* Unless noted, assume all other registers can be modified */
/* by this code.                                           */
/*                                     */
/*****/

```

```

procedure pat.l_OneOrMoreCset( cst:cset ); @nodisplay;

```

```

var

```

```

    cursor:    misc.pChar;    // Save last matched posn here.
    startPosn: misc.pChar;    // Save start of str here.
    endStr:    misc.pChar;    // End of string goes here.
    espSave:   dword;        // To clean stk after back trk.
    FailToSave: pat.FailRec;  // Save global FailTo value here.

```

```

begin l_OneOrMoreCset;

```

```

    // If some routine after this one fails and transfers
    // control via backtracking to this code, the stack
    // will be a mess. So save esp so we can clean up
    // the stack if backtracking is necessary.

```

```

    mov( esp, espSave );

```

```

    // Save the pointer to the start of the string
    // to match. This is used as a "fence" value
    // to prevent backtracking past the start of
    // the string if things go really wrong.

```

```

    mov( esi, startPosn );
    mov( esi, ebx );

```

```

    // Save pointer to end of string to match.
    // This is needed to restore this value when
    // backtracking occurs. If we're already
    // beyond the end of the chars to test, then
    // fail right away.

```

```

mov( edi, endStr );
if( esi >= edi ) then

    pat._fail_( pat.FailTo );

endif;

// Okay, this is a deferred version.  So match as
// few characters as possible.  For this routine,
// that means match exactly one character.

xor( eax, eax );
mov( [esi], al );           // Get the char to test.
bt( eax, (type dword cst)); // See if in cst.
if( @nc ) then

    pat._fail_( pat.FailTo );

endif;

// So we can easily back track, save a pointer
// to the next character.

inc( esi );
mov( esi, cursor );

// Save existing FailTo address and
// point FailTo at our back tracking code,
// then transfer control to the success
// address (jump to our return address).

pat._success_( FailToSave, FailToHere );


// If someone after us fails and invokes
// backtracking, control is transfered to
// this point.  First, we need to restore
// ESP to clean up the junk on the stack.
// Then we need to advance one character
// and see if the next char would match.

FailToHere:

    mov( espSave, esp );           // Clean up stack.

    mov( cursor, esi );           // Get last posn we matched.
    mov( endStr, edi );           // Restore to original value.

    // If we've exceeded the maximum limit on the string,
    // or the character is not in cst, then fail.

    xor( eax, eax );
    if
    {
        cmp( esi, edi );
        jae true;
        mov( [esi], al );
        bt( eax, (type dword cst) );
    }

```

```

        jc false;
    }

    // Need to restore FailTo address because it
    // currently points at us.  We want to jump
    // to the correct location.

    mov( startPosn, esi );
    mov( FailToSave.ebpSave, pat.FailTo.ebpSave );
    mov( FailToSave.jumpAdrs, pat.FailTo.jumpAdrs );
    pat._fail_( pat.FailTo );

endif;

// If we drop down here, there is at least one
// character left in the string that we've
// matched, so call the next matching routine
// (by jumping to the return address) to continue
// the pattern match.

mov( startPosn, ebx );
inc( esi );                // Advanced to next posn
mov( esi, cursor );        // save for backtracking,
mov( [ebp+4], eax );       // and call next routine.
mov( [ebp], ebp );
jmp( eax );

end l_OneOrMoreCset;

end patterns;

```





## 26 Random Number Generator Module (rand.hhf)

The rand.hhf header file contains definitions for HLA's random number generators. These functions provide a variety of pseudo-random number generators and support routines.

### 26.1 The Random Module

To use the random number generator functions in your application, you will need to include one of the following statements at the beginning of your HLA application:

```
#include( "rand.hhf" )
or
#include( "stdlib.hhf" )
```

### 26.2 The Random Number Generators

#### **procedure rand.randomize;**

This function "randomizes" the seed used by the random number generators. If you call *rand.randomize*, the random number generators should begin generating a sequence starting at a random point in the normal sequence put out by the random number generator.

The randomization function is based on the number of CPU clock cycles that have occurred since the CPU was last powered up. This function uses the Pentium's RDTSC instruction, hence you should only call this function on machines that have this instruction available (Intel Pentium and later as well as other manufacturer's CPUs that have this instruction).

Because of the nature of the RDTSC instruction, you should not call *rand.randomize* frequently or you will compromise the quality of the random numbers (indeed, it's generally not a good idea to "randomize" a random number generator more than once per program invocation). Similarly, you should avoid calling this function from a fixed script after power-on since that may also degrade the quality of the randomization. (These two suggestions are only important to those who are extremely concerned about the quality of the randomness of the generated numbers).

HLA high-level calling sequence example:

```
rand.randomize();
```

HLA low-level calling sequence example:

```
call rand.randomize;
```

#### **procedure rand.uniform; @returns( "eax" );**

This function generates a new random number on each call. This function returns a new 32-bit value in the EAX register on each call (bit 31 is randomly set, you may choose to interpret this value as a signed or unsigned integer). This function generates uniformly-distributed random numbers.

This function uses an algorithm from Knuth's *The Art of Computer Programming* for details (and limitations) on this type of random number generator.

HLA high-level calling sequence example:

```
rand.uniform();
mov( eax, randomValue );
```

HLA low-level calling sequence example:

```
call rand.uniform;
mov( eax, randomValue );
```

```
procedure rand.urange( startRange:int32; endRange:int32 ); @returns( "eax" );
```

This function generates a uniformly distributed random number in the range "*startRange..endRange*" (inclusive). This function generates its random numbers using the *rand.uniform* function. This function returns the value in the EAX register. This uses the same random-number generator algorithm that *rand.uniform* uses.

HLA high-level calling sequence example:

```
rand.urange( minValue, maxValue );
mov( eax, randomValue );
```

HLA low-level calling sequence example:

```
push( minValue );
push( maxValue );
call rand.urange;
mov( eax, randomValue );
```

```
procedure rand.random; @returns( "eax" );
```

This function generates a uniformly distributed random number using a linear congruential random number generator. This function returns a new 32-bit value in the EAX register on each call (bit 31 is randomly set, you may choose to interpret this value as a signed or unsigned integer). This function generates uniformly-distributed random numbers.

See Knuth's *The Art of Computer Programming* for details (and limitations) on linear congruential random number generators.

HLA high-level calling sequence example:

```
rand.random();
mov( eax, randomValue );
```

HLA low-level calling sequence example:

```
call rand.random;
mov( eax, randomValue );
```

```
procedure rand.range( startRange:int32; endRange:int32 ); @returns( "eax" );
```

This function generates a uniformly distributed random number in the range "*startRange..endRange*" (inclusive) using a linear congruential random number generator. This function returns the value in the EAX register. This uses the same random-number generator algorithm that *rand.random* uses.

HLA high-level calling sequence example:

```
rand.range( minValue, maxValue );
mov( eax, randomValue );
```

HLA low-level calling sequence example:

```

push( minValue );
push( maxValue );
call rand.range;
mov( eax, randomValue );

```

```

iterator rand.deal( count:uns32 );

```

The *rand.deal* iterator returns a sequence of *count* unique randomly arranged values in the range *0..count-1*. Therefore, it returns all values in the range *0..count-1*, but in a random order.

Since *rand.deal* is an iterator, you must only use it within a FOREACH loop, e.g.,

```

foreach deal( 52 ) do

    << EAX contains a value in the range 0..51 here>>

endfor;

```

This function uses the *rand.uniform* function to randomly arrange the values.



## 27 RPC: The Remote Procedure Calls Library

The HLA Standard Library Remote Procedure Call (RPC) module provides a very easy-to-use mechanism for creating and calling remote procedures. A remote procedure is one that exists in a different process space (possibly even on a different machine). One program (the *local* process) invokes a procedure within a second program (the *remote* process). To a very large degree, the HLA stdlib RPC module lets you define and call remote procedures in a manner quite similar to defining and calling local procedures.

The HLA Stdlib *rpc.hhf* module includes a small compiler for a *mini-language* that processes remote procedure declarations. Once you declare a remote procedure prototype in this mini-language (and the declarations will prove familiar to HLA programmers), very little additional syntax is needed to support remote procedure calls.

### 27.1 Types of Remote Procedures

The HLA stdlib RPC module supports three types of remote procedures: *synchronous*, *bisynchronous*, and *asynchronous*. These three procedure types allow you to balance flexibility and ease-of-use when writing and calling remote procedures.

A synchronous remote procedure comes the closest to resembling local procedure call semantics. When you call a synchronous remote procedure from some local process, that call will not return until the remote procedure returns from the call (and the remote procedure server sends an indication of this to the local process). Synchronous remote procedures are the easiest to understand because they mimic, to a fair degree, the semantics of a local procedure call.

An asynchronous remote procedure call does not block while waiting for the remote procedure to complete execution. Once the local *proxy function*<sup>1</sup> marshalls<sup>2</sup> all the parameters and transmits them to the remote procedure server, the procedure immediately returns and program execution continues in the local process. Asynchronous procedure calls are generally much faster (on the local machine) because the asynchronous procedure call doesn't have to wait for the remote procedure to finish (and send an acknowledgement over the network, which can be slow).

Bisynchronous procedures are a combination of synchronous and asynchronous procedures. The initial call is asynchronous, but there is a special function that you can call that will block the local process until the remote procedure returns (and the remote server acknowledges this). Bisynchronous procedures have one additional benefit not present in the other two remote procedure forms: you can return function results from a remote procedure when using bisynchronous calls.

### 27.2 Remote Procedure Call (RPC) Protocol

The HLA stdlib RPC module supports a peer-to-peer remote procedure call protocol. This protocol is built upon network sockets and is OS independent (that is, the HLA stdlib RPC code does not rely upon any particular RPC facilities provided by a specific operating system). The protocol operates in a dual client/server architecture. There are two processes involved in a remote procedure call; the server process is where the remote procedures reside, the client process calls the remote procedures. The protocol is a dual client/server model because the client process also runs a background server to receive responses and acknowledgements from the server. For this reason, a remote procedure server can only service one client at a time (effectively making this a peer-to-peer architecture).

Note that the RPC server code runs in a separate thread on the server machine. This means that in addition to linking in the socket library, code that uses remote procedures must also link in the threads module (and, therefore, you must supply the "-thread" command-line parameter on all modules you compile).

This architecture forces some important limitations. First of all, as just noticed, a set of remote procedures under the control of a remote server can only be called by one client until the client gives up the network connection to the server. This limitation isn't as bad as it seems because it's perfectly possible to run multiple RPC servers within the same process. So one program could be running multiple RPC servers and serving multiple clients, though this is not expected to be a common usage of the HLA RPC module.

The RPC server serializes all calls made to it. Although the local process can run multiple threads, with each thread making remote procedure calls, the calls are received one at a time in the server and queued up for execution. The server retrieves on request at a time, executes the associated code, acknowledges the call (if

---

1. A proxy function is a local procedure that is stand-in for the remote procedure. Calling a proxy function uses the same syntax as the actual function; the proxy collects all the arguments (marshalls them) and transmits the arguments to the remote procedure server for execution.

2. Marshall, in this context, means to collect the arguments into a single package.

synchronous or bisynchronous), and the fetches and processes the next call request. One advantage to this approach is that the remote procedures can assume that they are not executing concurrently (with respect to one another). If concurrent execution is an absolute necessity, the solution is the same as for multiple clients -- just run multiple RPC servers in the remote process and make the calls to different RPC servers; those calls will run concurrently (and it will be your responsibility to synchronize access to shared objects).

On the local process side, the HLA RPC module serializes calls to a single remote procedure. That is, if two different threads call the same remote procedure call (at the same time), the RPC library will block one of the two procedure calls until the other returns. For asynchronous RPCs, this is only as long as it takes to marshall the parameters and transmit them to the remote server. For synchronous and bisynchronous calls, the second call is blocked until the first one is acknowledged. In theory, this could also be fixed by running multiple servers (calling the same code on the remote server), but this would create some synchronization problems you'd have to handle manually.

Although a major goal (indeed, the main goal) of the HLA stdlib RPC module is to minimize the semantic differences between local and remote procedure calls, there are some things that are impractical to simulate in a remote procedure call and return. For example, changes to register values on the remote machine will not be reflected in the registers in the local process (in theory, it might be possible to do this, but it would be too expensive to do so). Likewise, because the local and remote processes run in different address spaces (indeed, usually on different machines) passing parameters by reference (or by any other scheme other than by value) is impractical.

## 27.3 The RPC Declaration Language (RDL)

The primary goal of the HLA stdlib RPC module is to minimize the (syntactical) differences between remote and local procedure calls. Largely, this is accomplished via the RPC Declaration Language (RDL). You make the RDL available to your programs by including the *rpc.hhf* header file at an appropriate spot at the beginning of your source file (e.g., near the other *#include* statements in your program). This makes all the macros, on which the RDL is built, available to your program. For example:

```
program t;
#includeOnce( "stdlib.hhf" )
#includeOnce( "rpc.hhf" )
.
.
.
end t;
```

Note that *rpc.hhf* automatically includes the *sockets.hhf* and *threads.hhf* header files. As noted earlier, this means that you will need to supply the *-thread* command-line parameter when compiling all HLA files in the project. If you attempt to compile a source file that includes *rpc.hhf* without supplying the *-thread* command-line parameter, HLA will emit a warning complaining about its absence.

The RDL uses the following basic syntax (these statements must appear after you've included the *rpc.hhf* header file):

```
remoteProcedures( classNamePrefix )

    << Synchronous, asynchronous, and bisynchronous procedure declarations >>

endRemoteProcedures
```

The *classNamePrefix* argument specifies the client and server names. The *remoteProcedures* statement will generate a couple of classes named *classNamePrefix\_server\_t* and *classNamePrefix\_client\_t*. It will also create a couple of instance variables named *classNamePrefix\_server* and *classNamePrefix\_client* (note the lack of a "t" on these instance variable names). These class/objects are singletons; that is, there is only one instance variable associated with each class (it doesn't really make any sense to have multiple objects of these classes). You will sometimes refer to these singleton instance variables (and the typenames) within your programs, but most of the time you will not use them; most of the work they do will be behind your back.

Within the body of the *remoteProcedures..endRemoteProcedures* statement are the actual remote procedure declarations. These take one of the following three forms:

```

sync( <<procedure declaration>> )
async( <<procedure declaration>> )
bisync( <<procedure declaration>> { :optional_return_type } )

```

A *<<procedure declaration>>* looks very similar to an HLA procedure declaration. It takes one of the following two forms:

```

procedureID
procedureID( <<optional parameter list>> )

```

*procedureID* is a (unique) HLA identifier that specifies the remote procedure's name. The *<<optional parameter list>>* item is a list of (pass by value) HLA parameter declarations whose syntax is identical to a normal formal parameter list declaration (except only implicit pass by reference is allowed). Here are some examples of legal remote procedure declarations:

```

sync( proc1 )                      // No parameters
bisync( proc2( i:int32; j:uns32; k:real32 ) ) // Three arguments
async( proc3( x:string ) )         // One argument

```

There are some severe limitations on the types of arguments you can pass to a remote procedure. In particular, you can only pass arguments of the following types:

- byte, boolean, char, uns8, int8
- word, uns16, int16
- dword, uns32, int32, real32
- qword, uns64, int64, real64
- tbyte, real80
- lword, cset, uns128, int128
- string
- blob\_t (note: blob.t, blob.blob, and blob.blob\_t will *not* work)

In particular, note that you cannot pass composite types such as arrays or records. We'll see how to manually pass these types of arguments a little later in this document. You cannot pass pointers or thunks to a remote procedure (which wouldn't make any sense because the remote procedure executes in a different address space). Of course, you could cast a pointer or thunk to a dword or qword, but keep in mind that the addresses in the remote procedure won't match those in the local procedure, so doing so will likely result in crashing the system or producing weird results. Remember, all arguments must be passed by value to a remote procedure.

Bisynchronous remote procedures support an optional return result. In the example immediately above, the bisynchronous procedure `proc2` does not have a return result associated with it. The following examples demonstrate some `bisync` declarations with a return result:

```

bisync( func1( x:real32 ):real64 )
bisync( rmtFileName:string )
bisync( rmtAppend( s1:string; s2:string ):uns64 )

```

The set of return types are also limited to the basic types listed earlier. Again, returning pointers and thunks is a no-no and there is no direct support for returning a composite data type. We'll look at ways to return composite data structures later in this document.

Here is a complete `remoteProcedures...endRemoteProcedures` statement using the declarations given thus far::

```

remoteProcedures( classNamePrefix )

    sync( proc1 )
    bisync( proc2( i:int32; j:uns32; k:real32 ) )
    async( proc3( x:string ) )
    bisync( func1( x:real32 ):real64 )
    bisync( rmtFileName:string )
    bisync( rmtAppend( s1:string; s2:string ):uns64 )

```

```
endRemoteProcedures
```

Generally, you're going to place the `remoteProcedures.endRemoteProcedures` statement within its own header file. For example, the code above might appear as part of the *myRPCs.hhf* header file:

```
// myRPCs.hhf:

#ifdef( !@defined( rpc_hhf ))
  ?rpc_hhf := true;

  #includeOnce( "rpc.hhf" )    // May as well do this here

  remoteProcedures( classNamePrefix )

    sync( proc1 )
    bisync( proc2( i:int32; j:uns32; k:real32 ) )
    async( proc3( x:string ) )
    bisync( func1( x:real32 ):real64 )
    bisync( rmtFileName:string )
    bisync( rmtAppend( s1:string; s2:string ):uns64 )

  endRemoteProcedures

#endif
```

The reason you'll want this in a header file is because you have to include this code in two files: in your local client program and in the remote server program.

If you stick anything besides a `sync`, `bisync`, or `async` statement between the `remoteProcedures` and `endRemoteProcedures` clauses, The RDL will ignore those statements and pass them straight through to HLA. Because the RDL statements do not directly emit any code, this is equivalent to placing those statements immediately before or after the `remoteProcedures.endRemoteProcedures` statement. Generally, it's bad style to do this; if you have a good reason for wanting to emit some code (or declarations) at that point, you can just as easily place them before the `remoteProcedures` statement.

As just noted, the RDL statements do not directly emit any code. Instead, they (effectively) create two header files containing the code they produce. These two header files are named *rpc\_client\_implementation.hhf* and *rpc\_server\_implementation.hhf*, respectively.<sup>3</sup> You should include the *rpc\_client\_implementation.hhf* header file in the local client code (generally immediately after including the header file containing the RDL code) and you should include the *rpc\_server\_implementation.hhf* header file in the server source file, e.g.,:

```
program rpcServer;
#include( "stdlib.hhf" )
#includeOnce( "myRDLcode.hhf" )
#includeOnce( "rpc_server_implementation.hhf" )
.
.
.
<< remainder of RPC server program >>

end rpcServer;
```

And for the client:

---

3. Technically, the RDL statements do not create these header files. The RDL statements create two text strings that these header files expand. You could expand those strings directly rather than including the header files, but there are technical reasons (dealing with error reporting) for expanding them in these header files.



```

program rpcClient;
#include( "stdlib.hhf" )
#includeOnce( "myRDLcode.hhf" )
#includeOnce( "rpc_client_implementation.hhf" )
.
.
.
<< remainder of RPC client program >>

end rpcClient;

```

Note that you cannot include both RPC implementation header files in the same source file. This will result in duplicate symbol definition errors.

## 27.4 RPC Preliminaries

The HLA `stdlib` RPC module works across TCP/IP using the HLA sockets library. This means that you need two things in order to establish communication between a local client and a remote procedure server: an IP address (of the server) and a pair of unused socket port numbers. It is possible to run the remote procedure server on the same machine as the local client (this is actually a very useful configuration for testing purposes), though you will typically run the remote server on a separate machine. After all, if you're always calling the remote procedures on the same machine as the local client, it would probably be more efficient to use threads rather than remote procedure calls. The HLA examples download contains an example of an RPC client and server that both run on the same machine. They call the `sock.hostAdrs` function to get the IP address of the machine they are running on. In general, however, it is your responsibility to determine the IP address of the machine running the RPC server.

Note that the server doesn't need to be explicitly given the IP address of the client. When the client connects with the server, it automatically transmits its own IP address to the server, so the server can use that address to connect make the second client/server connection back to the client.

The RPC protocol uses two consecutive socket port numbers. You supply only one port number and the RPC code computes the second value by adding 1 to the value you supply. One port is used for communication between the local client and the RPC server, the other port number is used for communication between the server and the client. Generally, any port number greater than 8000 that doesn't conflict with other software you're currently running is fine. The example in the HLA examples download uses port number 9998 (and 9999), but this value was chosen at random and has no official meaning.

Because the RPC system uses sockets, your client and server programs must initialize the socket subsystem by calling the HLA `stdlib` procedure `sock.socketInit`. After that, you're ready to create the client or server object and get things running.

## 27.5 RPC Clients

As noted earlier, the `remoteProcedures..endRemoteProcedures` statement will automatically construct two classes (one for the server module and one for the client module). You control the name of the classes via the argument you supply to `remoteProcedures`, e.g.,:

```

remoteProcedures( myRPC )

    sync( proc1 )
    bisync( proc2( i:int32; j:uns32; k:real32 ) )
    async( proc3( x:string ) )
    bisync( func1( x:real32 ):real64 )
    bisync( rmtFileName:string )
    bisync( rmtAppend( s1:string; s2:string ):uns64 )

endRemoteProcedures

```

This declaration, in the client file (that includes `rpc_client_implementation.hhf`), creates a class named `myRPC_client_t`. It also creates the VMT for that class and a single `static` object instance of that class named `myRPC_client`. Like any statically allocated class object, you'll need to initialize that object before you use it. You accomplish this by calling the `myRPC_client.create` constructor for the class. This constructor has two arguments: the IP address of the remote server and the port number (first of two) that the server will be listening on for the client.

The constructor actually does a lot more than simply initialize the class object. It actually connects to the server. It will not return until a connection is made with the server process. Upon return from the constructor, you're ready to make some remote procedure calls. Here is a typical invocation of the constructor:

```
myRPC_client.create( ipAdrs, $9998 );
stdout.put( "Connected to RPC server" nl );
```

When you compile the RDL statements (in the *myRDLcode.hhf* file in the earlier examples), the RDL compiler generates a considerable amount of code and places it in the *rpc\_client\_implementation.hhf* header file. For all remote procedure types, the RDL compiler will generate a local *proxy function* that has the same name and calling sequence as your declared remote procedures. For the current example, you wind up with procedure prototypes like the following:

```
procedure proc1; external;
procedure proc2( i:int32; j:uint32; k:real32 ); external;
procedure proc3( x:string );
procedure func1( x:real32 ); external;
procedure rmtFileName; external;
procedure rmtAppend( s1:string; s2:string ); external;
```

The RDL compiler actually writes the code for these functions (you compile this code into your client program by including the *rpc\_client\_implementation.hhf* header file). These proxy functions collect (marshall up) any arguments and send a packet to the remote server identifying the procedure to run and supplying the arguments for that call.

On the client side, once you've initialized the *myRPC\_client* object, you're free to call these procedures just as though they were local procedures.

For synchronous and asynchronous remote procedures (*proc1* and *proc3* in this example), that's all there is to it. You call the proxy procedure and the code executes on the remote server. For synchronous procedures (e.g., *proc1*), the call doesn't return until the code completes execution on the server (and the server notifies the client of the completion). For asynchronous procedure calls (e.g., *proc3*), the call returns immediately after the proxy procedure marshalls the arguments and ships them off to the remote server; asynchronous calls do not wait for the completion of the code on the remote server.

Note that the RDL compiler will create an additional client-side procedure in addition to the proxy functions for synchronous procedures. This procedure will have the same base name as the proxy procedure with *"\_return"* appended to the name (e.g., *proc1\_return* in the current example). This procedure is for internal use only. You must never call this procedure; doing so may make the system unreliable. Note that the RDL compiler does not generate any extra procedures for asynchronous procedure declarations. On the client, the proxy function is the only code the RDL compiler generates for an asynchronous procedure.

Bisynchronous procedures are considerably more complex than synchronous or asynchronous procedures. Like asynchronous and synchronous procedures, the RDL compiler will emit a proxy function that marshalls all the arguments and ships them over to the remote server for execution. Like asynchronous procedures, this proxy function will immediately return after it transmits the arguments to the remote server; it will not wait for the completion of the remote procedure call. Like synchronous procedures, the RDL compiler will create a *"\_return"* procedure (which you must never call) that the server remotely invokes on the client when the remote procedure completes execution.

The RDL compiler generates one additional client-side function for bisynchronous procedures: a *"\_waitForReturn"* function (that has the proxy name prepended to it, e.g., *func1\_waitForReturn*). The *"\_waitForReturn"* procedure serves two purposes: it delays the client program until the remote procedure completes execution and it provides a mechanism for retrieving a bisynchronous function return result.

Unlike the *"\_return"* function, a client-side application must call the *"\_waitForReturn"* function at some point after calling the proxy function. **This is absolutely required! If you fail to call the corresponding *"\_waitForReturn"* function after a bisynchronous procedure call, you will deadlock (hang) the system if you make a second call to that same proxy function.** Calling the proxy function enters a critical section associated with the remote procedure and calling the corresponding *"\_waitForReturn"* function leaves that critical section. If you attempt to call the proxy function twice without an intervening *"\_waitForReturn"* call, you will attempt to reenter the critical section (in the same thread) and this may produce deadlock. Moral of the story, always be careful when using bisynchronous procedures and ensure that you call the *"\_waitForReturn"* procedure as quickly as is reasonable.

Bisynchronous procedures are quite useful for continuing to do some work while you're waiting for the remote procedure to finish execution (and, generally the slower activity, waiting for the client and server to communicate the call and its completion between themselves). You can call a bisynchronous procedure, do some

work (that doesn't depend on the completion of that procedure), and then call the `"_waitForReturn"` procedure when you need to synchronize the execution of the local client code with the remote server.

Another reason for bisynchronous procedure calls is to return function results from a remote procedure. If you specify a return type when declaring a bisynchronous procedure, e.g.,:

```
bisync( func1( x:real32 ):real64 )
```

Then the RDL compiler will generate a `func1_waitForReturn` procedure with the following prototype:

```
procedure func1_waitForReturn( var rtn:real64 );
```

Note that this has a single pass-by-reference argument whose type is the same as the `bisync` return type. Calling such a typed `"_waitForReturn"` function will store the remote function's return result in the variable you pass as the procedure's parameter.

String and `blob_t` types are special cases. Consider the `rmtFileName` function from the earlier examples and its `"_waitForReturn"` function:

```
bisync( rmtFileName:string )

// Return function prototype:

rmtFileName_waitForReturn( var rtn:string );
```

The client-side remote procedure code automatically allocates storage for a string (or blob) result on the heap and stores a pointer to that new string (or blob) in the variable you pass to `rmtFileName_waitForReturn`. **This function does not store the string or blob data in the object you pass; it overwrites the variable's data pointer with the pointer to the new string or blob on the heap.**

These semantics have two ramifications in your program. First of all, if the string variable you pass to `rmtFileName_waitForReturn` already points at a string on the heap and you do not have another copy of that pointer laying around, then you will have a memory leak upon return from `rmtFileName_waitForReturn` when the original pointer is overwritten by the new one. Second, because the remote procedure call system allocates the storage for the return string on the heap, it is your responsibility to free this storage when you are done using the string (by calling `str.free` for strings or `blob.free` for blobs).

When you are done using the remote procedure server, you can shut down the connection by calling the destructor for the client class object. For example, if you're using the name `myRPC` (as in the earlier examples that described the constructor), you can shut down the system with the following call:

```
myRPC.destroy();
```

Generally, it's a good idea to wait a second or two after calling the destructor to give the remote system time to shut down before you kill the socket connection (this is optional, it's not strictly required). You can do this with an HLA `stdlib` call such as `os.sleep(1)`;

## 27.6 RPC Servers

The server side of a remote procedure call client/server pair is slightly more complicated than the client side. This is largely because in addition to initializing (and destroying) the server object, you've actually got to supply the remote procedures that the client will be calling. As on the client side, synchronous and asynchronous procedures are fairly easy to understand -- they look and behave much like local procedures; bisynchronous procedures, on the other hand, introduce some additional complexities because of return results and their bisynchronous nature.

Like the client side, the first thing you'll find of interest in a remote procedure server program is the inclusion of the RDL source code and the associated implementation file. Continuing the example from the previous system, here's a reminder of what the basic program file will look like:

```
program rpcServer;
#include( "stdlib.hhf" )
#includeOnce( "myRDLcode.hhf" )
#includeOnce( "rpc_server_implementation.hhf" )
.
```

```

        .
        .
    << remainder of RPC server program >>

end rpcServer;

```

The structure of the main program is going to be somewhat different from the client. This is because the server exists primarily to provide remote procedure call services to the client. The client, on the other hand, exists to solve some application problems and remote procedure calls might only be a tiny part of the work that takes place on the client. On the server, however, the work revolves totally around providing that remote procedure call service.<sup>4</sup> A typical main program in an RPC server will consist of two calls: one call to the server's class constructor and one call to its destructor.

When you compile the RDL code (i.e., the `remoteProcedures..endRemoteProcedures` statement), the RDL compiler generates a class and an object instance variable specifically for the server application. Assuming you've specified the name `myRPC` in the `remoteProcedures` statement, the class name will be `myRPC_server_t` and the object instance will be `myRPC_server`. The constructor for this new class has the following calling syntax:

```

myRPC_server.create
(
    basePortNumber,
    &serverConnectedProcedure,
    timeoutThunk
);

```

The `basePortNumber` argument is the port number (first of two). This must match the value you use on the client side. Remember that the RPC server and client use two consecutive port numbers. So if you specify 9998 as the port number, the system will actually use ports 9998 and 9999.

The `serverConnectedProcedure` argument is the name of a parameterless procedure that the RPC server will call when the client successfully connects when the server and the server, in turn, successfully connects with the acknowledgement server on the client side. There is one requirement for this procedure: it must call the "connected" method of the `*_server_t` class created by the RDL compiler. Assuming the use of the `myRPC` argument to `remoteProcedures`, here's what the minimal `serverConnectedProcedure` will look like:

```

procedure serverConnectedProcedure;
begin serverConnectedProcedure;

    myRPC_server.connected();

end serverConnectedProcedure;

```

Though this is the minimal amount of work this procedure should do, there are some minor embellishments that are useful. Here is a typical example (taken from the RPC example in the HLA examples download):

```

static
    quitServer :boolean;

procedure serverConnected;
begin serverConnected;

    stdout.put( "Client connected with server" nl );

    // Start the real server code:

    myRPC_server.connected();

```

---

4. Technically, there is no reason you couldn't start up a thread in the server program and run the remote procedure call server in that thread while the main thread does other work, but generally your system will be more robust if you limit the activities of the remote procedure call server to just handling remote procedure calls.

```

stdout.put( "Client disconnected from server" nl );
mov( true, quitServer );

end serverConnected;

```

Printing "Client connected with server" and "Client disconnected from server" are obvious modifications (note that when `myRPC_server.connected` returns, the client and server have disconnected). The reason for setting the global variable `quitServer` to true will become apparent in a few moments.

The last argument to the server class constructor is a "timeout thunk." If you're unfamiliar with thunks, just note that they are a fancy form of a procedure that you can declare in-line in some other code. The cool thing about thunk parameters (as opposed to procedures you pass by address as a parameter) is that you can encode the thunk statements in-line in the argument list of a procedure call.

The constructor calls the `timeoutThunk` on a periodic basis (the interval is set by the thunk itself). On entry into the server's constructor thunk, `EAX` points at an `hla.timeval` variable that the thunk can use to change the timeout period. On return, `EAX` contains false if the constructor is to continue waiting for the server to connect, `EAX` should contain true if a timeout has occurred and you want to return from the constructor without connecting. Here's a typical constructor call with an in-line thunk:

```

mov( false, quitServer );
myRPC_server.create
(
    $9998,
    &serverConnected,
    thunk
    #{
        // On entry to thunk, EAX contains the address of the timeout
        // variable. Set this as desired for the timeout (0.1 second,
        // in this case).

        if( eax <> NULL ) then

            // Timeout is 0.1 seconds while waiting for
            // connection:

            mov( 100_000, (type hla.timeval [eax]).tv_usec );

        endif;
        movzx( quitServer, eax );
    }#
);

```

Notice that this thunk returns the value of `quitServer` in `EAX`. Because this code sequence also initializes `quitServer` with false (and no other code in the thunk ever sets it to true), this code will never time out. The `myRPC` constructor will continually call this thunk at 0.1 second (100,000 microsecond) intervals until it connects with a client.

When the client calls its destructor, the client sends a message to the server telling it to disconnect. Whenever a client disconnects from the server, the server reenters the loop waiting for another client to connect with it. However, if you look back at the `serverConnected` code given earlier, it sets the `quitServer` global variable to true, so the first time the constructor reenters the thunk, the thunk returns true in `EAX` and the server quits execution.

Upon returning from the constructor, it's a good idea for the server to delay a couple of seconds before calling the class destructor method (`myRPC_server.destroy`) and quitting the program:

```

// Short delay to allow all transmissions to complete before we bail:

os.sleep( 2 );

my_server.destroy();

```

All that remains in the server module is to write the actual remote procedures that the client will call. As for the client side, asynchronous and synchronous procedures are relatively straight-forward and look just like local procedure calls. For example:

```
procedure proc1;
begin proc1;

    stdout.put( "Client called synchronous procedure proc1" nl );

end proc1;

procedure proc3( x:string );
begin proc3;

    stdout.put( "Client called asynchronous procedure proc3, x=", x, nl );
    str.free( x );

end proc3;
```

This is relatively straight-forward stuff. About this only issue of which you must be aware (and `proc3` demonstrates) is that when you pass string values as parameters, the RPC code allocates storage for the string object on the heap and it is your responsibility to free that storage when you are done using the string's data.

Behind your back, the RDL compiler generates some additional functions for all procedures (synchronous, asynchronous, and bisynchronous). These procedures have the same name as the primary procedure with the addition of a " `marshall`" suffix. The `marshall` procedures read the arguments (if any) from the network, save them into local variables, and then call the actual user-written procedure with these arguments. It should go without saying that user-written code should never call these marshalling procedures; the marshalling procedures expect data coming across the network and they must only be called from the RPC server in response to an RPC request from the client.

As on the client side, bisynchronous procedures are slightly more complex than asynchronous and synchronous procedures. Let's first consider the `proc2` example from earlier in this document:

```
bisync( proc2( i:int32; j:uns32; k:real32 ) )
```

Here is a simple implementation of `proc2`:

```
procedure proc2( i:int32; j:uns32; k:real32 );
begin proc2;

    stdout.put( "proc2 was called, i=", i, ", j=", j, ", k=", k, nl );
    proc2_return();

end proc2;
```

The important thing to notice in this code is the call to the `proc2_return` procedure. This is an RPC compiler generated procedure that notifies the client when the procedure is done executing. Generally, you must call this procedure at the end of a bisynchronous procedure (that's the logical place to call this procedure, though there is nothing stopping you from calling it earlier if you have a good reason to do so). **You must call this function exactly once in every bisynchronous procedure.** If you don't call it, the client will hang up waiting for the bisynchronous procedure call to finish when it calls the corresponding `proc2_waitForReturn` function.

Now consider an example of a bisynchronous remote procedure that has a function return result:

```
//    bisync( rmtFileName:string )

procedure rmtFileName;
begin rmtFileName;

    // Code to compute the filename and produce the string
    // value "fileName"
```

```

    rmtFileName_return( fileName );

end rmtFileName;

```

Whenever a bisynchronous procedure returns a function result, the corresponding `"*_return"` function will require a single (pass by value) parameter whose type matches the return type. Note that the return function will completely transmit the data before it returns. Therefore, you are free to destroy the object (e.g., free the storage associated with the `fileName` string variable) upon return from the return function.

## 27.7 Passing Large Objects Between the Client and Server

The existing RDL does not support passing composite data types (other than strings and blobs) to and from remote procedures. In this section you'll see how to overcome this limitation. You can pass large data types between a client and server, however, you'll have to manually pass that data yourself rather than relying on RDL compiler generated code to do the job for you.

The RPC2 project in the HLA examples download (which will be reproduced here) demonstrates how to pass large chunks of data between a client and an RPC server. Passing a large data structure to the RPC server is the easiest to understand, so we'll start with that explanation.

There are two ways to pass large chunks of data: using explicit networking calls or moving the data to a blob and transmitting the blob. We'll start with a discussion of using explicit networking calls.

Although the RPC protocol passes fixed amounts of data between the client and the server, this protocol is built on top of the HLA sockets library, that allows you to transmit an arbitrary amount of data from one network node to another. Inside the classes that the RDL constructs are two HLA socket objects: a `client_t` object and a `server_t` object, named `client` and `server`, respectively. As long as you are careful, you can use these `client` and `server` objects to communicate data from the client application to the RPC server application.

Before discussing how to pass a block of data from the client to the server using explicit networking calls, an important warning is worth mentioning: the RPC server process has a very strict data transmission and receipt protocol. You will be injecting bytes into the (client) transmission stream and intercepting bytes in the (client) reception stream. The code you write on the server and client sides to handling these extra bytes must transmit and receive the exact number of bytes your code on the other side of the network is expecting. The HLA stdlib RPC module is not fault tolerant with respect to recovering from a protocol mish-mash (TCP/IP guarantees correct delivery, it doesn't make sense to replicate the error checking in the RPC code). If you transmit `M` bytes to the server but your code on the server only reads `(M-N)` of those bytes (or attempts to read `M+N` bytes), then the RPC protocol will be out of sync and unexpected results (usually a crash) will happen shortly thereafter. Moral of the story: if you transmit `M` extra bytes from one network node to the other, make sure the other reads those `M` bytes (and no more).

If you want to transmit a large block of bytes from the client to the RPC server using explicit networking calls, the first thing you're going to need to do is to create a bisynchronous procedure to handle the transmission. This *has* to be a bisynchronous procedure. The reason for a bisynchronous procedure is because the code you write on the server side will need to read the extra bytes you send it before the RPC server attempts to read another command from the network socket. Bisynchronous procedures give your application complete control over this process on the server as well as complete control over the acknowledgement receipt on the client side. Synchronous procedures won't work because the client-side call will wait for an acknowledgement before you get control back (so you won't be able to send the array data until after the server has acknowledged the call and has already started waiting for a new command). Asynchronous procedures could work, but it's easier to mess up the RPC protocol with asynchronous procedures, so bisynchronous procedures make a better choice.

The first place to start is with the data structure you want to pass from the client to the server. Generally, it's a very good idea to place a type definition for this data type in the same header file that contains your RDL code (that is, the `remoteProcedures...endRemoteProcedures` statement). For this section's example, we're going to pass a `dword` array with 16 elements (that is, a 64-byte object). Here's the data type declaration that appears in the `sc.hhf` header file in the RPC2 example:

```

type
    array_t      :uns32[16];

```

By convention (a convention I'm creating as I write this, as this is the first example of this process ever written), the remote procedure on the RPC server will expect a single `dword` parameter that will hold the size of the object. Technically, no such parameter is necessary because the data type (`array_t` in this example) is visible on both the client and server sides and both sides can easily compute the size of the object using the `@size( array_t )` compile-time language function invocation. However, by explicitly passing the size, you

can run a consistency check on the server side just to verify that there aren't any problems. Here's what a typical RDL statement for a procedure with a large array argument might look like:

```
bisync( passBig( size:dword )    )
```

On the client side, you're going to need to write a separate function (separate from the one the RDL compiler generates for `passBig`) that you can use to pass the array to the server. This new procedure will call `passBig` to get the server side process running and then it will transmit the array data to the server before waiting for the server to return an acknowledgement. In the RPC2 example, I've called this procedure `passArray`. Here is the code for `passArray`:

```
procedure passArray( var a:array_t );
var
    b:blob_t;
begin passArray;

    push( esi );
    push( edi );

    passBig( @size( array_t ) );
    lea( esi, myRPC_client.client );
    (type client_t [esi]).write( val a, @size( array_t ) );
    passBig_waitForReturn();

    pop( edi );
    pop( esi );

end passArray;
```

The call to `passBig` in this procedure kicks off the data transmission process. The `passBig` procedure on the server side will accept the size argument and then the user-written code in that procedure will wait for the arrival of the data. When the server receives all the data, it will send an acknowledgement back and the `passBig_waitForReturn` function will return and the client will continue execution.

Upon entry into the `passBig` procedure on the server, we've already received the `size` parameter value and the client has probably sent the bulk data on its way as well. Therefore, one of the first things we've got to do is to read that block of data from the network socket. Once we've read all the network data, we have to call the `passBig_return` procedure to send an acknowledgement to the client that the procedure is done executing. Here's the complete code for the `passBig` procedure:

```
procedure passBig( size:uns32 );
var
    anArray :array_t;

begin passBig;

    assert( size = @size( array_t ) );
    lea( esi, myRPC_server.server );
    (type server_t [esi]).read( anArray, @size( array_t ) );
    stdout.put( "Received anArray:" nl );
    for( mov( 0, ecx ); ecx < @elements( array_t ); inc( ecx ) ) do

        stdout.put( "anArray[" , (type uns32 ecx) , "]=", anArray[ecx*4], nl );

    endfor;
    passBig_return();

end passBig;
```

Notice how the call to `passBig_return` occurs after we've read all the data from the client.



The second way, and arguably the standard way, to pass a large data type from the client to the RPC server is by passing a `blob_t` argument. On the client side, you would create a `blob_t` object, store the large data object into the blob, and then remotely call the desired procedure passing the blob argument.

Before demonstrating how this is done, you should be aware of an important fact: if you intend to pass a blob object between the client and the server you must use the `blob_t` data type because the RDL compiler recognizes only this blob type. Although `blob.t`, `blob.blob_t`, and `blob.blob` are usually synonyms for `blob_t`, the RDL compiler doesn't recognize these synonyms (for syntactical reasons).

One advantage of passing large data types as blobs is that you can create synchronous, bisynchronous, and asynchronous remote procedures without any trouble. In this example, we'll use a synchronous procedure declaration. From the *sc.hhf* header file (in the RPC2 project in the HLA examples downloads) you'll find the following RDL declaration for the `passBigb` procedure:

```
sync( passBigb( b:blob_t ) )
```

In the client source file, you'll need to write a procedure that will collect your large object's data together and place that information into a blob (assuming the large object isn't a blob to begin with). Then you will call the remote procedure and pass that blob as an argument. When you're done with the blob you've created, you will typically free its storage before returning. Here's the implementation of `passArray2` from the RPC2 project:

```
procedure passArray2( var a:array_t );
var
    b:blob_t;
begin passArray2;

    blob.alloc( @size( array_t ) );
    mov( eax, b );
    blob.write( b, val a, @size( array_t ) );
    passBigb( b );
    blob.free( b );

end passArray2;
```

First, this code allocates a blob object large enough to hold the data object to pass to the remote procedure. Then it writes the data (from the array in this example) to the blob. Then it calls `passBigb` to pass the array data, encapsulated in the blob, over to the server. Finally, it deletes the storage associated with the blob.

On the server side, the `passBigb` procedure receives the blob data via a `blob_t`-typed parameter. This procedure simply has to read the data out of the blob and place it in the local array data object. Whenever you pass a blob object to a remote procedure, that remote procedure is required to free the storage associated with that blob (same as the situation for strings). Here's the server side code from the RPC2 example that demonstrates this activity:

```
procedure passBigb( b:blob_t );
var
    bArray :array_t;

begin passBigb;

    blob.length( b );
    assert( eax = @size( array_t ) );

    lea( eax, bArray );
    blob.read( b, [eax], @size( array_t ) );
    stdout.put( nl "Received bArray:" nl );
    for( mov( 0, ecx ); ecx < @elements( array_t ); inc( ecx ) ) do

        stdout.put( "bArray[", (type uns32 ecx), "]= ", bArray[ecx*4], nl );

    endfor;
```

```

        blob.free( b );

    end passBigb;

```

To return a large data object as a function result there is only one mechanism available: pack the data into a `blob_t` object and return that blob to the client (and unpack the blob when the client receives it). As is the case for all remote procedure that return a value, you must use a bisynchronous procedure to achieve this. Here's the RDL declaration from the RPC2 example program for a `rtnBig` procedure that returns a 16-element array object:

```

    bisync( rtnBig:blob_t )

```

On the server side, you've got to create a blob, move the large data object into the blob, return the blob as the remote function result, and then free up the blob you've created. Here's the same code from the RPC2 *server.hla* source file:

```

procedure rtnBig;
var
    b:blob_t;
    a:array_t;

begin rtnBig;

    // Create an array to return:

    mov( @elements( array_t ), eax );
    for( mov( 0, ecx ); ecx < @elements( array_t ); inc( ecx ) do

        mov( eax, a[ecx*4] );
        dec( eax );

    endfor;

    // Copy the array into a blob:

    blob.alloc( @size( array_t ) );
    mov( eax, b );
    blob.write( b, a, @size( array_t ) );

    // Return the array:

    rtnBig_return( b );

    // Free the blob:

    blob.free( b );

end rtnBig;

```

On the client side, after receiving the blob (which the RPC library module has already allocated storage on the stack for), you've got to unpack the blob data into the desired data structure and then free the storage associated with the blob. Here's the client code to achieve this:

```

procedure returnArray( var a:array_t );
var
    b:blob_t;

begin returnArray;

```

```
    rtnBig();  
    rtnBig_waitForReturn( b );  
    blob.read( b, val a, @size( array_t ));  
    blob.free( b );  
  
end returnArray;
```



## 28 Sockets Module (sockets.hhf)

The HLA Standard Library provides two mechanisms that support network communications via sockets: a low-level library (whose functions appear in the *sock* namespace) and a higher-level socket class that makes it almost trivial to create client/server applications. The low-level *sock* library provides a thin veneer over the low-level OS calls (largely to make the calls portable across all OSes). Those wanting to write networking applications using traditional BSD-style socket calls should consider using the *sock* module rather than the low-level OS calls.

**Note:** HLA also provides access to certain low-level OS API primitives that directly access the native OS' networking code. Please see the discussions of the native OS functions for more details on such low-level network access.

### 28.1 The SOCK Module

The *sock* namespace in the HLA standard library contains a fair set of functions that behave largely like the original BSD socket functions (that most OSes support). These functions, by and large, work just like the low-level socket calls that most OSes support. Largely, these function smooth out some data type differences between various OSes (e.g., the definition of the *fd\_set\_t* data type) so that function calls (and the data types of their arguments) are consistent across all OSes, regardless of the underlying data types a particular OS might use.

### 28.2 Socket Initialization and Cleanup

Before using any socket functions, you must first call the *sock.socketInit* function to initialize the socket library. When you are done using sockets in an application, you must call *sock.socketCleanup* to free system resources and shut down the socket system. These are HLA standard library functions that are not particularly related to the underlying OS socket API. You must call these function before any other socket operations and when you're done using the sockets.

**sock.socketInit;**

This function initializes the socket library for the HLA standard library. You must call this function exactly one in any application that makes other low-level socket calls (before making those calls). Note that this function may increment an internal reference counter, so make sure you make a corresponding call to *sock.socketCleanup* before your application terminates.

```
sock.socketInit();
```

**sock.socketCleanup;**

This function undoes the effects of *sock.socketInit* and frees up any system resources reserved by *sock.socketInit*. You must call it exactly once when your application is done using sockets.

```
sock.socketCleanup();
```

### 28.3 Generic Socket Functions

A few functions in the *sock* namespace provide conversions on socket metadata. These functions include *sock.a\_adrsToStr*, *sock.adrsToStr*, and *sock.strToAdrs*.

```
sock.a_adrsToStr( a:bigEndianDW ); @returns( "eax" );
sock.adrsToStr( a:bigEndianDW; s:string );
```

These functions take a dword parameter in *network byte order (big endian form)* and convert the address to the form "ddd.ddd.ddd.ddd" (where each "ddd" represents exactly three decimal digits). The *sock.a\_adrsToStr* function allocates storage for the 15-character string on the heap and returns a pointer to the new string in the EAX register. The *sock.adrsToStr* function stores the string result into the string object passed as the second argument (*s*). The *sock.adrsToStr* function will raise an exception if *s* doesn't have sufficient storage to hold a 15-character string.

```

sock.a_adrsToStr( $01020304 );           // Produces "004.003.002.001"
sock.adrsToStr( $04030201, s );          // Stores "001.002.003.004" into s

```

```
sock.strToAdrs( s:string ); @returns( "eax" );
```

This function takes a string parameter of the form "ddd.ddd.ddd.ddd" (where each "ddd" represents exactly three decimal digits) and converts it to a double word *in network byte order (big endian form)* and returns this value in the EAX register. This function raises an exception if there is a conversion error.

```
sock.strToAdrs( "001.002.003.004" ); // Produces $04030201 in EAX
```

## 28.4 Low-Level BSD-Style Socket Functions

The functions in this category correspond to the Berkeley (BSD) sockets functions. You should not assume that the data types passed to these functions are identical to those in BSD sockets. Some data types have been changed in order to make the HLA sockets module compatible across all the OSes that the HLA stdlib supports. It is the responsibility of all of these functions to do any necessary conversion prior to calling the OS-level socket API functions.

This documentation will not describe the functionality for each of these functions. See a discussion of the BSD sockets API (on the internet) for more details. If you are unfamiliar with low-level socket calls, you should either use the HLA standard library socket classes (which simplify network programming) or pick up a good book on making networking calls via the BSD sockets API. You can also look at the source code for the socket server and client classes in the HLA standard library for examples of these calls.

```

sock.accept
(
    s                :dword;
    var addr         :sock.sockaddr;
    var addrlen      :sock.socklen_t
);

```

The argument *s* is a socket that has been created with *sock.socket*, bound to an address with *sock.bind*, and is listening for connections after a *sock.listen* call. The *sock.accept* function extracts the first connection request on the queue of pending connections, creates a new socket with the same properties of *s* and allocates a new file descriptor for the socket. If no pending connections are present on the queue, and the socket is not marked as non-blocking, *sock.accept* blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, *sock.accept* returns an error as described below. The accepted socket may not be used to accept more connections. The original socket *s* remains open.

The argument *addr* is a result parameter that is filled in with the address of the connecting entity, as known to the communications layer. The exact format of the *addr* parameter is determined by the domain in which the communication is occurring. The *addrlen* is a value-result parameter; it should initially contain the amount of space pointed to by *addr*; on return it will contain the actual length (in bytes) of the address returned. This call is used with connection-based socket types, currently with *sock.SOCK\_STREAM*.

It is possible to *sock.select* a socket for the purposes of doing a *sock.accept* by selecting it for read.

This function raises an *ex.SocketError* exception if any error occurs.

```

sock.bind
(
    sockfd           :dword;
    var addr         :sock.sockaddr;
    addrlen          :socklen_t
);

```

*sock.bind* assigns a name (that is, an IP address) to an unnamed socket. When a socket is created with *sock.socket* it exists in a name space (address family) but has no name (IP address) assigned. *sock.bind* requests that name (IP address) be assigned to the socket.

This function raises an *ex.SocketError* exception if any error occurs.

```

sock.connect
(
    var      s          :dword;
            serv_addr    :sockaddr;
            addrlen      :socklen_t
);

```

The parameter *s* is a socket. If it is of type *sock.SOCK\_DGRAM*, this call specifies the peer with which the socket is to be associated; this address is that to which datagrams are to be sent, and the only address from which datagrams are to be received. If the socket is of type *sock.SOCK\_STREAM*, this call attempts to make a connection to another socket. The other socket is specified by name (i.e., IP address), which is an address in the communications space of the socket. Each communications space interprets the name parameter in its own way. Generally, stream sockets may successfully connect only once; datagram sockets may use *sock.connect* multiple times to change their association. Datagram sockets may dissolve the association by connecting to an invalid address, such as a null address or an address with the address family set to *sock.AF\_UNSPEC*.

This function raises an *ex.SocketError* exception if any error occurs.

```

sock.close( s:dowrd );

```

*sock.close* closes the socket whose handle is specified by the *s* descriptor passed as a parameter.

This function raises an *ex.SocketError* exception if any error occurs.

```

sock.listen
(
    backlog    s      :dword;
              :dword
);

```

To accept connections, a socket is first created with *sock.socket*, a willingness to accept incoming connections and a queue limit for incoming connections are specified with *sock.listen*, and then the connections are accepted with *sock.accept*. The *sock.listen* call applies only to sockets of type *sock.SOCK\_STREAM* or *sock.SOCK\_SEQPACKET*.

The backlog parameter defines the maximum length the queue of pending connections may grow to. If a connection request arrives with the queue full the client may receive an error, or, if the underlying protocol supports retransmission, the request may be ignored so that retries may succeed.

This function raises an *ex.SocketError* exception if any error occurs.

```

sock.recv
(
    var      s          :dword;
            buf         :var;
            len         :dword;
            flags       :dword
);  @returns( "eax" );

```

```

sock.recvfrom
(
    var      s          :dword;
            buf         :var;
            len         :dword;
            flags       :dword;
            var      from      :sockaddr;

```

```

    var      fromlen      :socklen_t
); @returns( "eax" );

```

*sock.recvfrom* is used to receive messages from a socket, and may be used to receive data on a socket whether or not it is connection oriented.

If *from* is non-nil, and the socket is not connection-oriented, the source address of the message is filled in. *Fromlen* is a value-result parameter, initialized to the size of the buffer associated with *from*, and modified on return to indicate the actual size of the address stored there.

The *sock.recv* call is normally used only on a connected socket (see *sock.connect*) and is identical to *sock.recvfrom* with a *NJLL from* parameter.

On successful completion, both routines return the number of message bytes read in the EAX register. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from (see *sock.socket*). Note that if these functions read fewer bytes from the socket than specified by the *len* parameter, these functions do not raise an end-of-file exception (as is common for the socket class input routines).

The receive calls normally return any data available, up to the requested amount, rather than waiting for receipt of the full amount requested; this behavior is affected by the socket-level options *sock.SO\_RCVLOWAT* and *sock.SO\_RCVTIMEO* described in *sock.getsockopt*.

The *sock.select* call may be used to determine when more data arrives.

The flags argument to a *sock.recv* call is formed by or'ing one or more of the values:

```

    sock.MSG_OOB    process out-of-band data
    sock.MSG_PEEK    peek at incoming message
    sock.MSG_WAITALL wait for full request or error

```

The *sock.MSG\_OOB* flag requests receipt of out-of-band data that would not be received in the normal data stream. Some protocols place expedited data at the head of the normal data queue, and thus this flag cannot be used with such protocols. The *sock.MSG\_PEEK* flag causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, a subsequent receive call will return the same data. The *sock.MSG\_WAITALL* flag requests that the operation block until the full request is satisfied. However, the call may still return less data than requested if a signal is caught, an error or disconnect occurs, or the next data to be received is of a different type than that returned.

This function raises an *ex.SocketError* exception if any error occurs.

```

sock.select
(
    nfd      :dword;
    var readSet :sock.fd_set_t;
    var writeSet :sock.fd_set_t;
    var exceptSet :sock.fd_set_t;
    var timeout :sock.timeval
); @returns( "eax" );

```

The *sock.select* function examines the I/O descriptor sets whose addresses are passed in *readfds*, *writefds*, and *exceptfds* to see if some of their descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. The first *nfd* descriptors are checked in each set; i.e., the descriptors from 0 through *nfd-1* in the descriptor sets are examined. (Example: If you have set two file descriptors "4" and "17", *nfd*s should not be "2", but rather "17 + 1" or "18".) On return, *sock.select* replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation.

Select() returns the total number of ready descriptors in all the sets in the EAX register.

Note that the *sock.fd\_set\_t* data type may not be equivalent to the *fd\_set* data type used by the underlying operating system. In particular, you should not assume that this is a bit map. The function may choose to ignore the *nfd*s parameter (which is present for historical reasons), but you should still set it up properly.

If *timeout* is a non-nil pointer, it specifies a maximum interval to wait for the selection to complete. If *timeout* is a NULL pointer, the select blocks indefinitely. To effect a poll, the *timeout* argument should be non-NULL, pointing to a zero-valued *sock.timeval* structure. *Timeout* is not changed by *sock.select*, and may be reused on subsequent calls, however it is good style to re-initialize it before each invocation of *sock.select*.

Any of *readfds*, *writefds*, and *exceptfds* may be given as nil pointers if no descriptors are of interest.

The *sock.select* function returns the number of ready descriptors that are contained in the descriptor sets. If the time limit expires, *sock.select* returns 0. If *sock.select* raises an exception, the descriptor sets will be unmodified.

This function raises an *ex.SocketError* exception if any error occurs.



```

sock.send
(
    s          :dword;
    var buf    :var;
    len        :dword;
    flags      :dword    // MSG_* constants
); @returns( "eax" );

sock.sendto
(
    s          :dword;
    var buf    :var;
    len        :dword;
    flags      :dword    // MSG_* constants
    var _to    :sock.sockaddr;
    tolen      :sock.socklen_t
); @returns( "eax" );

```

The *sock.send* and *sock.sendto* functions are used to transmit a message to another socket. The *sock.send* function may be used only when the socket is in a connected state, while *sock.sendto* may be used at any time.

The address of the target is given by *to* with *tolen* specifying its size. The length of the message is given by *len*. If the message is too long to pass atomically through the underlying protocol, the function raises an *ex.SocketError* exception and the message is not transmitted.

No indication of failure to deliver is implicit in a *sock.send* call.

If no messages space is available at the socket to hold the message to be transmitted, then *sock.send* normally blocks, unless the socket has been placed in non-blocking I/O mode. The *sock.select* call may be used to determine when it is possible to send more data.

The flags parameter may include one or more of the following:

```

sock.MSG_OOB      /* process out-of-band data */
sock.MSG_DONTROUTE/* bypass routing, use direct interface */

```

The flag *sock.MSG\_OOB* is used to send ``out-of-band" data on sockets that support this notion (e.g. *sock.SOCK\_STREAM*); the underlying protocol must also support ``out-of-band" data. *sock.MSG\_DONTROUTE* is usually used only by diagnostic or routing programs.

The call returns the number of characters sent in the EAX register.

This function raises an *ex.SocketError* exception if any error occurs.

```

sock.socket(int domain, int type, int protocol);@returns( "eax" );

```

*sock.socket* creates an endpoint for communication and returns a descriptor. The domain parameter specifies a communications domain within which communication will take place; this selects the protocol family which should be used. These families are defined in the include file *sockets.hhf*.

The currently understood formats are

```

sock.AF_UNIX      (UNIX internal protocols),
sock.AF_INET      (ARPA Internet protocols),

```

The socket has the indicated type, which specifies the semantics of communication. Currently defined types are:

```

sock.SOCK_STREAM
sock.SOCK_DGRAM

```

A *sock.SOCK\_STREAM* type provides sequenced, reliable, two-way connection based byte streams. An out-of-band data transmission mechanism may be supported. A *sock.SOCK\_DGRAM* socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length).

The protocol specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner. The protocol number to use is particular to the communication domain in which communication is to take place.

Sockets of type *sock.SOCK\_STREAM* are full-duplex byte streams, similar to pipes. A stream socket must be in a connected state before any data may be sent or received on it. A connection to another socket is created with a *sock.connect* call. Once connected, data may be transferred using some variant of the *sock.send* and *sock.recv* calls. When a session has been completed a *sock.close* may be performed. Out-of-band data may also be transmitted as described in *sock.send* and received as described in *sock.recv*.

The communications protocols used to implement a *sock.SOCK\_STREAM* stream insure that data is not lost or duplicated. If a piece of data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error by raising an *ex.SocketError* exception. The protocols optionally keep sockets "warm" by forcing transmissions roughly every minute in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for an extended period (e.g. 5 minutes). An *ex.SocketError* exception is raised if a process sends on a broken stream; this causes naive processes, which do not handle the exception, to exit.

The operation of sockets is controlled by socket level options. *sock.setsockopt* and *sock.getsockopt* are used to set and get options, respectively.

If the call is successful, the return value (in EAX) is a descriptor referencing the socket.

This function raises an *ex.SocketError* exception if any error occurs.

```
sock.setsockopt
(
    s           :dword;
    level       :dword;
    optname     :dword;
    var optval   :var;
    optlen      :socklen_t
); @returns( "eax" );
```

```
sock.getsockopt
(
    s           :dword;
    level       :dword;
    optname     :dword;
    var optval   :var;
    optlen      :socklen_t
); @returns( "eax" );
```

*sock.getsockopt* and *sock.setsockopt* manipulate the options associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost "socket" level.

When manipulating socket options the level at which the option resides and the name of the option must be specified. To manipulate options at the socket level, level is specified as *sock.SOL\_SOCKET*. To manipulate options at any other level the protocol number of the appropriate protocol controlling the option is supplied. For example, to indicate that an option is to be interpreted by the TCP protocol, level should be set to the protocol number of TCP.

The parameters *optval* and *optlen* are used to access option values for *sock.setsockopt*. For *sock.getsockopt* they identify a buffer in which the value for the requested option(s) are to be returned. For *sock.getsockopt*, *optlen* is a value-result parameter, initially containing the size of the buffer pointed to by *optval*, and modified on return to indicate the actual size of the value returned. If no option value is to be supplied or returned, *optval* may be NULL.

*Optname* and any specified options are passed uninterpreted to the appropriate protocol module for interpretation. The include file *sockets.hhf* contains definitions for socket level options, described below.

Most socket-level options utilize a dword parameter for *optval*. For *sock.setsockopt*, the parameter should be non-zero to enable a Boolean option, or zero if the option is to be disabled. *sock.SO\_LINGER* uses a record *sock.linger* parameter, defined in *socket.hhf*, which specifies the desired state of the option and the linger interval (see below). *sock.SO\_SNDTIMEO* and *sock.SO\_RCVTIMEO* use a *sock.timeval* parameter, defined in *sockets.hhf*.

The following options are recognized at the socket level. Except as noted, each may be examined with *sock.getsockopt* and set with *sock.setsockopt*.

<i>sock.SO_DEBUG</i>	enables recording of debugging information
<i>sock.SO_REUSEADDR</i>	enables local address reuse
<i>sock.SO_REUSEPORT</i>	enables duplicate address and port bindings
<i>sock.SO_KEEPAIVE</i>	enables keep connections alive
<i>sock.SO_DONTROUTE</i>	enables routing bypass for outgoing messages
<i>sock.SO_LINGER</i>	linger on close if data present
<i>sock.SO_BROADCAST</i>	enables permission to transmit broadcast messages
<i>sock.SO_OOBINLINE</i>	enables reception of out-of-band data in band
<i>sock.SO_SNDBUF</i>	set buffer size for output
<i>sock.SO_RCVBUF</i>	set buffer size for input
<i>sock.SO_SNDLOWAT</i>	set minimum count for output
<i>sock.SO_RCVLOWAT</i>	set minimum count for input
<i>sock.SO_SNDTIMEO</i>	set timeout value for output
<i>sock.SO_RCVTIMEO</i>	set timeout value for input
<i>sock.SO_TYPE</i>	get the type of the socket (get only)
<i>sock.SO_ERROR</i>	get and clear error on the socket (get only)
<i>sock.SO_NOSIGPIPE</i>	do not generate SIGPIPE, instead return EPIPE

*sock.SO\_DEBUG* enables debugging in the underlying protocol modules. *sock.SO\_REUSEADDR* indicates that the rules used in validating addresses supplied in a *sock.bind* call should allow reuse of local addresses. *sock.SO\_REUSEPORT* allows completely duplicate bindings by multiple processes if they all set *sock.SO\_REUSEPORT* before binding the port. This option permits multiple instances of a program to each receive UDP/IP multicast or broadcast datagrams destined for the bound port. *sock.SO\_KEEPAIVE* enables the periodic transmission of messages on a connected socket. Should the connected party fail to respond to these messages, the connection is considered broken and processes using the socket are notified via an *ex.SocketError* exception when attempting to send data. *sock.SO\_DONTROUTE* indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

*sock.SO\_LINGER* controls the action taken when unsent messages are queued on socket and a *sock.close* is performed. If the socket promises reliable delivery of data and *sock.SO\_LINGER* is set, the system will block the process on the close attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, termed the linger interval, is specified in the *sock.setsockopt* call when *sock.SO\_LINGER* is requested). If *sock.SO\_LINGER* is disabled and a close is issued, the system will process the close in a manner that allows the process to continue as quickly as possible.

The option *sock.SO\_BROADCAST* requests permission to send broadcast datagrams on the socket. Broadcast was a privileged operation in earlier versions of the socket system. With protocols that support out-of-band data, the *sock.SO\_OOBINLINE* option requests that out-of-band data be placed in the normal data input queue as received; it will then be accessible with *sock.recv* without the *sock.MSG\_OOB* flag. Some protocols always behave as if this option is set. *sock.SO\_SNDBUF* and *sock.SO\_RCVBUF* are options to adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections, or may be decreased to limit the possible backlog of incoming data. The system places an absolute limit on these values.

*sock.SO\_SNDLOWAT* is an option to set the minimum count for output operations. Most output operations process all of the data supplied by the call, delivering data to the protocol for transmission and blocking as necessary for flow control. Nonblocking output operations will process as much data as permitted subject to flow control without blocking, but will process no data if flow control does not allow the smaller of the low water mark value or the entire request to be processed. A *sock.select* operation testing the ability to write to a socket will return true only if the low water mark amount could be processed. The default value for *sock.SO\_SNDLOWAT* is set to a convenient size for network efficiency, often 1024. *sock.SO\_RCVLOWAT* is an option to set the minimum count for input operations. In general, receive calls will block until any (non-zero) amount of data is received, then return with the smaller of the amount available or the amount requested. The default value for *sock.SO\_RCVLOWAT* is 1. If *sock.SO\_RCVLOWAT* is set to a larger value, blocking receive calls normally wait until they have received the smaller of the low water mark value or the requested amount. Receive calls may still return less than the low water mark if an error occurs, a signal is caught, or the type of data next in the receive queue is different than that returned.

*sock.SO\_SNDBTIMEO* is an option to set a timeout value for output operations. It accepts a struct timeval parameter with the number of seconds and microseconds used to limit waits for output operations to complete. If a send operation has blocked for this much time, it returns with a partial count or raises an exception if no data were sent. In the current implementation, this timer is restarted each time additional data are delivered to the protocol, implying that the limit applies to output portions ranging in size from the low water mark to the high water mark for output. *sock.SO\_RCVTIMEO* is an option to set a timeout value for input operations. It accepts a record *sock.timeval* parameter with the number of seconds and microseconds used to limit waits for input operations to complete. In the current implementation, this timer is restarted each time additional data are received by the protocol, and thus the limit is in effect an inactivity timer. If a receive operation has been blocked for this much time without receiving additional data, it returns with a short count or with the error EWOULDBLOCK if no data were received. The struct

The *timeval* parameter must represent a positive time interval otherwise *sock.setsockopt* raises an *ex.SocketError* exception.

Finally, *sock.SO\_TYPE* and *sock.SO\_ERROR* are options used only with *sock.getsockopt*. *sock.SO\_TYPE* returns the type of the socket, such as *sock.SOCK\_STREAM*; it is useful for servers that inherit sockets on startup. *sock.SO\_ERROR* returns any pending error on the socket and clears the error status. It may be used to check for asynchronous errors on connected datagram sockets or for other asynchronous errors.

This function raises an *ex.SocketError* exception if any error occurs.

```
sock.setTimeout( s:dword; timeout: sock.timeval );
```

*sock.setTimeout* sets the timeout period for both transmission and reception on the socket specified by the socket descriptor *s*. Note: this function is a convenience function that calls *sock.setsockopt* to set the timeout periods. There is no equivalent *gettimeout* function; call *sock.getsockopt* if you need to retrieve one of the timeout periods. If you need to set the receive timeout period independently of the send timeout period, you will need to call *sock.setsockopt* to achieve this.

This function raises an *ex.SocketError* exception if any error occurs.

```
sock.gethostname( s:string );
```

The *sock.gethostname* makes a copy of (one of) the host name string(s) and stores this into the string variable you pass as an argument. Some systems can have more than one host name. In such a case, *sock.gethostname* returns one of the names (arbitrary choice).

This function raises an *ex.SocketError* exception if any socket-based error occurs. This function raises an *ex.StringOverflow* error if the hostname string is too long to fit in the storage allocated for the *s* argument. It can raise other exceptions if the value of *s* is bad.

```
sock.gethostbyname( s:string; var hstent:sock.hostent );
```

The *sock.gethostbyname* function fills in a *sock.hostent* data structure you pass by reference with host information based on the host name you pass as a string (*s*) to the function. Here is the current definition of the *sock.hostent* data structure (note that this is subject to change over time, so always use the *sock.hostent* type rather than manually creating this data structure yourself):

```
hostent:record

    h_name      :zstring;
    h_aliases   :dword;
    h_addrtype  :sock.sa_family_t;
    padding0    :word;
    h_length    :word;
    padding1    :word;
    h_addr_list :dword;

endrecord;
```

The *h\_aliases* field is a pointer to a sequence of dword addresses, terminated by a NULL address, each of which points at a zstring containing an alternate name for the host. You must not modify this array and you must not modify the strings its entries point at.

The *h\_addrtype* field contains the type of the address being returned. This is usually *sock.AF\_INET*.

The *h\_length* field contains the length, in bytes, of each address in the address list.

The *h\_addr\_list* is a pointer to an array of pointers to network addresses for the host. Note that these addresses are stored in network byte order (big endian) form. The list is terminated with a NULL pointer.

This function raises an *ex.SocketError* exception if any socket-based error occurs.

```
sock.gethostbyaddr
(
    var    addr        :var;
          len          :dword;
          _type        :dword;
    var    hstent:hostent
);
```

The *sock.gethostbyaddr* function fills in a *sock.hostent* data structure you pass by reference with host information based on the host whose address you pass to the function. The *addr* argument is the address of a network address data structure whose type is specified by the *type* argument (usually *sock.AF\_INET*) and the length of which is specified by the *len* argument. This function copies the host information to the *hstent* argument you pass by reference. The *h\_name* field of the *sock.hostent* data structure will contain the primary name of the host.

This function raises an *ex.SocketError* exception if any socket-based error occurs.

```
sock.getpeername
(
    s          :dword;
    var _name   :sock.sockaddr;
    var namelen :sock.socklen_t
); @returns( "eax" );
```

The *sock.getpeername* function returns the IP address (the "name") of the peer connected to the socket specified by the *s* socket descriptor. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *\_name*. On return it contains the actual size of the *\_name* returned (in bytes). The name is truncated if the buffer provided is too small. Note that the term "name" here refers to an IP address, not the name of the peer machine. This confusion is unfortunate, but that's the way the BSD sockets system was designed.

This function raises an *ex.SocketError* exception if any socket-based error occurs.

```
sock.getsockname
(
    s          :dword;
    var _name   :sock.sockaddr;
    var namelen :sock.socklen_t
); @returns( "eax" );
```

The *sock.getsockname* function returns the IP address (the "name") of the socket machine specified by the *s* socket descriptor. The *namelen* parameter should be initialized to indicate the amount of space pointed to by *\_name*. On return it contains the actual size of the *\_name* returned (in bytes). The name is truncated if the buffer provided is too small. Note that the term "name" here refers to an IP address, not the name of the peer machine. This confusion is unfortunate, but that's the way the BSD sockets system was designed.

This function raises an *ex.SocketError* exception if any socket-based error occurs.

```
sock.fd_zero( var fdset:sock.fd_set_t );
sock.fd_set( fd:dword; var fdset:sock.fd_set_t );
sock.fd_clr( fd:dword; var fdset:sock.fd_set_t );
sock.fd_isset( fd:dword; var fdset:sock.fd_set_t );@returns( "al" );
```

The *sock.fd\_\** functions manipulate "file descriptor sets" that the *sock.select* function uses. The *sock.fd\_zero* function creates the empty set and stores this into the *fdset* argument that you pass as by reference to the function. The *sock.fd\_set* function unions the file (socket) descriptor pass in *fd* into the set *fdset* that you pass by reference to the function. The *sock.fd\_clr* function removes (if preset) the file descriptor you pass in the *fd* argument from the set *fd\_set* that you pass by reference to the function. The *sock.fd\_isset* function checks for set membership. That is, it checks to see if the file descriptor specified by *fd* is present in the set *fdset* that you pass

by reference to the function; this function returns true/false in the AL register to denote presence/absence of the file descriptor in the set.

Note: always use these functions to manipulate socket descriptor sets. Do not assume that the HLA stdlib data structures match that of the underlying OS (they don't). Do not assume that the current implementation will always be used in future versions of the HLA stdlib. By using these functions, you can avoid future problems.

This function raises an *ex.SocketError* exception if any error occurs.

## 28.5 Socket Classes

**Warning:** *Don't forget that HLA objects modify the values in the ESI and EDI registers whenever you call a class procedure, method, or iterator. Do not leave any important values in either of these registers when making calls to the socket object functions. If the use of ESI and EDI is a problem for you, you might consider using the sock module that does not suffer from this problem.*

The HLA Standard Library provides an object-oriented network socket access mechanism implemented via the *baseSocket\_t*, *server\_t*, *client\_t*, *vBaseSocket\_t*, *vServer\_t*, and *vClient\_t* classes. As is typical for classes appearing in the HLA Standard Library, you can create customized versions of the generic socket classes, selecting which class functions are procedures or methods. This lets you choose between efficient static linking and virtual (overload) method capability on a function by function basis. Unless otherwise specified, this document will use the terms *socket class*, *server class*, and *client class* to describe the generic socket classes rather than the specific instance of these classes (which uses static linking for all functions).

The HLA Standard Library sockets module provide six predefined classes that simplify the use of sockets, particularly for client/server applications. There are three basic classes with two variants of each class (a static variant and a virtual variant). In HLA classes, there are three types of functions: (static) procedures, (dynamic) methods, and dynamic iterators. The only difference between a method and a procedure is how the program actually calls the function: the program calls procedures directly, it calls methods indirectly through an entry in the virtual method table (VMT). The system always calls object iterators indirectly through the VMT, so we will not consider them in this discussion. This section will discuss the impact of class procedures versus class methods in your programs.

Static procedure calls are very efficient, but you lose the benefits of inheritance and functional polymorphism when you define a function as a static procedure in a class. Methods, on the other hand, fully support polymorphic calls, but introduce some (in)efficiency issues. The following paragraphs describe some of the efficiency issues concerning the use of methods.

First of all, unlike static procedures, your program will link in all methods defined in your program *even if you don't explicitly call those methods*. Because the call is indirect, there really is no way for the assembler and linker to determine whether you've actually called the function, so it must assume that you do call it and links in the code for each method in the class. This can make your program a little larger because it may be including several socket class functions that you don't actually call.

The second efficiency issue concerning method calls is that they use the EDI register to make the indirect call (static procedure calls do not disturb the value in EDI). Therefore, you must ensure that EDI is free and available before making a virtual method call, or take the effort to preserve EDI's value across such a call.

A third, though extremely minor, efficiency issue concerning methods is that the class' VMT will need an extra entry in the virtual method table. As this is only four bytes per class (not per object), this probably isn't much of a concern.

The predefined *baseSocketClass\_t* and *vBaseSocketClass*, *server\_t* and *vServer\_t*, and *client\_t* and *vClient\_t* classes differ in how they define the functions appearing in the class types. The non-virtual types (without the 'v' prefix) generally use static procedures for all functions, the *virtual* types (with the 'v' prefix) use methods for all class functions. Therefore, the non-virtual socket object types will make direct calls to all the functions (and only link in the procedures you actually call); however, the non-virtual socket objects do not support function polymorphism in derived classes. The virtual socket types do support polymorphism for all the class methods, but whenever you use these data types you will link in all the methods (even if you don't call them all) and calls to these methods will require the use of the EDI register.

It is important to understand that *baseSocketClass\_t/vBaseSocketClass\_t*, *server\_t/vServer\_t*, and *client\_t/vClient\_t* pairs are two separate types. Neither is derived from the other. Nor are the two types in each pair compatible with one another. You should take care not to confuse objects of these two types if you're using both types in the same program.

The *baseSocketClass\_t* and *vBaseSocketClass\_t* types are base types intended for creating derived types (e.g., *server\_t/vServer\_t* and *client\_t/vClient\_t*); you would not normally use these two types in your programs, instead, you would use some type derived from these base classes (such as *server\_t/vServer\_t*, or *client\_t/vClient\_t*).

## 28.6 A Quick Note

The following sections do not include sample code demonstrating the calling sequences for a couple of reasons:

For high level calls, the syntax depends on the object name and type.

Low-level calling sequences don't appear here because it doesn't really make sense to make a low-level object invocation.

These functions are really intended for use by programmers experienced with HLA's Object-oriented assembly facilities.

For the same reasons, there are no stack diagrams for these function calls. If you want more information on making calls to HLA class methods and procedures, please consult the HLA documentation.

In the following function descriptions, the symbol <object> is used to specify a socket class object or a pointer to a socket class object. Wherever this document uses the name "socket", you may substitute (as appropriate) *server\_t*, *vServer\_t*, *client\_t*, *vClient\_t*, or any other socket class you've created by subclassing *baseSocketClass\_t* or *vBaseSocketClass\_t*.

## 28.7 Client/Server Applications Using the Socket Classes

The HLA sockets module makes creating client/server applications in assembly language almost trivial. The HLA Stdlib provides four classes for this purpose: *server\_t*, *vServer\_t*, *client\_t* and *vClient\_t*. The classes with the 'v' prefix use virtual methods for all functions, those without the 'v' prefix use static procedures. Generally, you'll use the static versions of these classes unless you need to create derived classes from them and overload some methods in these classes. Using the static classes produces more efficient code, but you lose the ability to overload the class functions (i.e., polymorphism is not possible with static classes). Note that you do not have to choose both virtual or static classes for your client and server applications. That is, one application can use a virtual class and the other can use a static class. The server and client applications are independent with respect to the choice of virtual or static classes. The examples in this document will use static classes because they don't require polymorphism, but you can easily substitute a virtual class into these examples, as needed.

## 28.8 A Simple Server Application

A server application is one that is running waiting for some other application (the client) to communicate with it. In particular, the server application must be running before the client application attempts to connect to it. Usually, the client and server applications run on separate computer systems on the network, though it is perfectly possible (and common for testing purposes) to run both applications on the same computer. The important thing to understand is that the server must be running before the client application begins because the client assumes that the server is available to provide services when it attempts to connect to the server.

To create a server application, you begin by declaring a *server\_t* variable, e.g.,

```
static
myServer :server_t;
```

The *server\_t* data type inherits all the functions from the *baseSocketClass\_t* (which this document will describe later) and it adds two methods you can call: *start* and *close*. Though the *server\_t* (and *baseSocketClass\_t*) type has some data fields, you should consider them private to the class and never access or modify them directly.

The *start* method has the following prototype:

```
method start
(
    adrs                :dword;
    port                :word;
    timeoutCallback     :thunk;
    connectionCallback  :procedure
); @returns( "eax" );
```

The first parameter is the IP address that this server will be listening for clients on. This is a 32-bit IP address *in little endian format*! This value is *not* in big endian (network byte order) form.

The second parameter is the port (socket) number that the server will listen on for a connection from a client. This is a 16-bit value *in little endian (not network byte order/big endian) format!* The combination of IP address and socket port number is what uniquely identifies a particular server.

The *timeoutCallback* parameter is a **thunk** that the *start* function calls before attempting to listen for a client and on each timeout period while listening for a client. For those unfamiliar with thunks, they are simply procedures embedded in other code; when called, the EBP register is initialized to the value of EBP in the surrounding code when the thunk was initialized. This means that the code in the thunk can access variables that are local to the code that the thunk is embedded in.

When the *start* method invokes the *timeoutCallback* thunk code, the EAX register contains the address of a *sock.timeval* object that controls the timeout period. On the first call, the *start* method has initialized this timeout to zero (which means infinite timeout period). If your thunk code does not change this value, then the server will wait indefinitely for a connection and will never again invoke the timeout thunk. If you would like to have your timeout thunk invoked on a periodic basis while the server is listening for a client connection (perhaps to update a progress bar or something like that to indicate the program is still operating), you should initialize the *sock.timeval* value pointed at by EAX. Before returning, the thunk should load a Boolean value (true or false, 1 or 0) into EAX to tell the *start* method whether it should quit. False/0 means "don't quit", true/1 means "quit" (and return to whomever called the *start* method).

Here is a typical thunk that sets up a one-second timeout period:

```
static
    timeout      :thunk;
    calls        :dword := 0;
    .
    .
    .
    thunk timeout :=
    #{
        // On entry to thunk, EAX contains the address of the timeout
        // variable. Set this as desired for the timeout (1 second,
        // in this case).

        mov( 1, (type sock.timeval [eax]).tv_sec );
        mov( 0, (type sock.timeval [eax]).tv_usec );

        // On successive calls, print a period to the stdout
        // to let an observer know that we're still listening
        // for a connection:

        cmp( calls, 0 );
        je dontPrintPeriod;

        stdout.putc( '.' );

        dontPrintPeriod:
        inc( calls );
        mov( 0, eax );    // Never quit

    }#;
```

After initializing the *timeout* thunk as shown above, you can pass the *timeout* thunk variable to the *start* method.

The fourth parameter to the *start* method is the address of a procedure that *start* will call when it connects with a client application over the network. This is a standard HLA procedure with no parameters. This procedure must preserve all registers it modifies. This *connectionCallback* procedure provides whatever service the client requires and generally operates in one of two modes:

The *connectionCallback* procedure directly provides whatever services the client requires and then returns. At that point, the server and client are disconnected and the server starts listening for a new client. In this mode, the server can provide services to only one client at a time.

The

*connectionCallback* procedure spawns a new process to handle the client's requests and then immediately returns to the *start* method. The server then begins listening for a new client connection while the spawned



process provides appropriate services for the client. This mode allows the server to provide services to multiple clients simultaneously.

Upon entry into the *connectionCallback* procedure, the EAX register contains a socket handle and the ESI register contains the address of a *server\_t* (or *vServer\_t*) object. You should save these values in local (not static!) variables. At the very least, you will need the object address in order to communicate with the client.

If you choose to spawn a new process to provide services to the client, you must make a copy of the server object address passed to you in the ESI register. This is because the value passed in ESI is the main object used by the *start* procedure and upon returning to *start* after you spawn the new process, *start* will write to the data fields of that object. This would be a disaster for the service process. You can avoid this problem by making a copy of the *server\_t* (or *vServer\_t*) using code like the following:

```

procedure connected;
    @nodisplay;
    @nostackalign;
    @noframe;
var
    handle      :dword;
    object      :pointer to server_t;
    newObject   :pointer to server_t;

begin connected;

    push( ebp );
    mov( esp, ebp );
    sub( _vars_, esp );

    pushad();
    pushfd();

    mov( eax, handle );
    mov( esi, object );
    server_t.create();
    mov( esi, newObject );

    // Duplicate the server object:

    mov( esi, edi );
    mov( object, esi );
    mov( @size( server_t ), ecx );
    cld();
    rep.movsb();

    /*****

    // Spawn process here and pass it the address of the new object
    // contained in newObject.

    // . . . . .

    *****/

    // Return to start, so it can handle other client requests

    popfd();
    popad();
    leave();
    ret( _parms_ );

end connected;
```

Note that the *server\_t start* method will never return unless the *timeout* thunk returns true in the EAX register. Therefore, if you elect to specify an indefinite timeout by storing zero into the *sock.timeval* object passed to the *timeout* thunk, the *start* method will never return to its caller.

Whenever a *connectionCallback* procedure (or the thread it spawns) finishes providing services for a connected client, it should call the server object's *close* method to disconnect the socket from the client and free up system resources. If you've allocated a new socket object (e.g., prior to spawning a new process), you should call the server *destroy* method and also free the storage associated with the socket object before terminating the spawned process.

## 28.9 A Simple Client Application

Setting up a client application is even easier than setting up a server application. You declare an object of type *client\_t* (or *vClient\_t*), invoke the object's *create* procedure, and then call the *connect* method to connect to a listening server. Here is the prototype for the *connect* method:

```
method connect
(
    adrs    :dword;
    port    :word
); @returns( "eax" );
```

The *adrs* parameter is the IP address (in little endian form). The *port* parameter is the socket port number (a 16-bit value, also in little endian form). When you call this function, it will attempt to connect to the server at the specified IP address and port number. If a server is not available or it refuses to connect to the client program, the *connect* method will raise an *ex.SocketError* exception. Note that the *connect* method will not wait for a server to become available. If it goes to the specify IP address and port number and there isn't a server application listening on that port, the *connect* method will raise an exception.

Once the *connect* method returns (without raising an exception), you can assume that the client and server are connected and communication between the two application may commence. When the client is done using the services of the server, it should call the *client\_t* (or *vClient\_t*) *close* method to disconnect itself from the server.

## 28.10 Client/Server Communication

Once a client establishes a connection to a server, the two applications may exchange data. A socket supports bidirectional data transfer; that is, the server can send data to the client and receive data from the client, and the client may send data to and receive data from the server.

The *server\_t/vServer\_t* and *client\_t/vClient\_t* classes are derived class that inherit all the information from the *baseSocketClass\_t/vBaseSocketClass\_t* classes. The *baseSocketClass\_t/vBaseSocketClass\_t* classes define all the procedures and methods that the server and client objects use to communicate via the socket. Without question, the most generic I/O functions (and the ones you will probably use most commonly) are the *read* and *write* functions. These methods/procedures use prototypes like the following:

```
method read( var buf:var; len:dword );
    @returns( "eax" );

method write( var buf:var; len:dword );
    @returns( "eax" );
```

The first parameter (*buf*) is the address of some block of memory, the second parameter (*len*) is the number of bytes to read or write at the address in memory. This function returns the number of bytes read or the number of bytes written in the EAX register.

All socket I/O communication is subject to a *timeout period*. The base socket class defines a (private) data field that specifies the timeout period in seconds and microseconds. You can use the socket class' *setTimeout* and *setTimeout2* functions to specify the timeout period (the default is zero, which means wait indefinitely). Should a timeout occur during a socket *read* or *write* call, the function will immediately return without completing the I/O operation. The EAX register will contain the actual number of bytes read or written; so you can check the return result to determine if the I/O operation was complete.

In addition to the generic *read* and *write* functions, the base socket classes provide a full set of formatted I/O functions similar to those provided by the *stdout*, *stdin*, *stderr*, and *fileio* modules. The following sections will describe the use of those functions.

## 28.11 General Socket Class Operations

The functions in this category let you initialize socket objects, access fields of the socket objects, and perform other conversion and housekeeping tasks.

```
<object>.create; @returns( "esi" );
server_t.create; @returns( "esi" );  [to create dynamic objects]
client_t.create; @returns( "esi" );  [to create dynamic objects]
```

The socket classes provide a *<socket>.create* constructor which you should always call before making use of a client or server variable. For socket variables (as opposed to socket pointer variables), you should call this routine specifying the name of the socket variable. For socket pointer variables, you should call this routine using the class name and store the pointer returned in EAX into your file variable. For example, to initialize the following two socket objects, you would use code like the following:

```
var
  MyClientSocket    : client_t;
  clientPtr         : pointer to vClient_t;
  .
  .
  .

MyClientSocket.create();

vclient_t.create();
mov( eax, clientPtr);
```

Note that the *vClient\_t.create* constructor simply initializes the virtual method table pointer and does other necessary internal initialization. The constructor does not open a socket or perform other socket-related activities.

```
<object>.destroy; @returns( "esi" );
server_t.destroy; @returns( "esi" );  [to create dynamic objects]
client_t.destroy; @returns( "esi" );  [to create dynamic objects]
```

The socket classes provide a *<socket>.destroy* destructor which you should always call when you're done using a socket. For example, when you are done working with the MyClientSocket and the clientPtr objects from the previous examples, you should execute the following code:

```
MyClientSocket.destroy();

clientPtr.destroy();
```

The socket destructor frees up system resources in use by an active socket. Note: some of these resources are system wide and may not be automatically reclaimed when your program terminates. Be sure you always call the destructor to prevent system resource leaks.

```
<object>.close;
server_t.close;
client_t.close;
```

This method closes a socket opened via *<object>.start* or *<object>.connect*.

```
MyClientSocket.close();
clientPtr.close();
```

Note that calling the *destroy* method/procedure does not close the socket. You must always call the *close* function before calling *destroy*. The difference between the two is that the *close* function tells the OS you're done using the socket, the *destroy* method deallocates resources associated with the HLA Standard Library.

```
<serverObject>.start
(
    adrs                :dword;
    port                :word;
    timeoutCallback     :thunk;
    connectionCallback  :procedure
);
```

This method starts a server that listens on IP address *adrs* and socket *port* number *port* for a connection from a client.

After setting up the server-side socket (but before checking for a client attempting to connect), this function calls the thunk specified by the *timeoutCallback* parameter. It passes the address of a *sock.timeval* variable to the thunk in the EAX register. The *timeoutCallback* thunk should set this *sock.timeval* variable to an appropriate timeout value (zero mean indefinite timeout). Generally, the timeout value is non-zero because you want to check the status of the listening socket on a periodic basis; further, the only way the *start* function ever returns to the caller is via a signal from the *timeoutCallback* thunk; Therefore, if you do not specify a timeout value (that is, if you specify an indefinite timeout period by writing zeros to the *sock.timeval* object), then *start* will have no way to terminate (other than by manually killing the process).

The *timeoutCallback* thunk passes true/false (1/0) back to the *start* function in the EAX register. If EAX contains true, then the *start* function returns to the caller and terminates listening for a client connection. If EAX contains zero upon return, then the *start* function continues to listen for a socket connection or until the timeout period expires.

The *start* function calls the procedure pointed at by the *connectionCallback* parameter whenever the server accepts a connection from a client. Upon entry into the *connectionCallback* procedure, ESI will contain the address of the server object (<serverObject>) and EAX will contain a copy of the new socket connection handle. At this point, most programs do one of two things: either the procedure pointed at by *connectionCall* provides all the services required by the client (during which the server will not accept any more client connections), or the procedure can spawn a new thread to provide those services and then immediately return (allowing the *start* function to handle additional client requests while the new thread provides any necessary services to the connected client).

The simplest case is to have the *connectionCallback* procedure provide all the services without spawning a new thread. In this case, the *connectionCallback* procedure would store the value in ESI into a *server\_t* (or *vServer\_t*) pointer variable and then use that pointer variable with all the I/O functions described in the following sections. When your code finishes, it simply returns to the *start* function and the server continues listening for new connections. Note, however, that while your server code is providing those services, the *start* function is suspended and your server will not accept any other client requests. This mode of operation is great for peer-to-peer type socket communications where only two network nodes communicate at one time.

If you want to allow your server to handle multiple client requests simultaneously, the situation is more complicated. First of all, you cannot simply store away the pointer held in the ESI register; instead, you have to make a copy of that object for use by the new thread and pass the address of this copy to the thread. After creating the copy, you should spawn a new thread (passing the address of the new object to the thread) and then reference the copy of the object within that thread. The reason for making a copy of the server object is because the server will modify that object on the next client connection. This would create problems for the current server thread.

Note that you don't actually have to create a new *server\_t* or *vServer\_t* object. The data server thread will only need a *baseSocketClass\_t* (or *vBaseSocketClass\_t*) object, so you can create one of those objects and then copy the pertinent fields from the *server\_t/vServer\_t* object to the *baseSocketClass\_t/vBaseSocketClass\_t* object (use the <object>.assign function to copy data from one *baseSocketClass\_t/vBaseSocketClass\_t* object into the current object).

Of course, don't forget that multithreaded applications have their own host of synchronization requirements. Also be aware that many of the stdlib functions are not (as this was being written) thread-safe, so be sure to protect stdlib calls with a mutex unless you are sure that the call you're making will function properly in a multithreaded environment.

```
MyServerSocket.start( $01020304, $1234, myTimeoutThunk, &connection );
```

```
<clientObject>.connect( IPAdrs:dword; port :word );
```

This method connects a client to a server.

```
MyClientSocket.connect( $01020304, $1234 );
```

This function attempts to connect a client to a server. The *IPAdrs* is the IP address of the server (in little endian form); *port* is the socket port number (also in little endian form). If the server is ready and willing to accept a socket connection, this function returns; if the server is not running, or is unwilling to connect, then this function raises an *ex.SocketError* exception.

```
<baseSocketClass_t>.assign( var src:baseSocketClass_t );
<vBaseSocketClass_t>.assign( var src:vBaseSocketClass_t );
```

These functions copy the pertinent data fields from the *src* operand to the current object. This is useful when creating a copy of a socket for use by a server thread (see the discussion in *start*'s description).

```
MySocket.assign( (type baseSocketClass_t [esi]) );
```

```
<object>.setTimeout( timeout:sock.timeval );
<object>.setTimeout2( tv_sec:dword; tv_usec:dword );
<object>.getTimeout( var timeout:sock.timeval );
```

These functions get and set the internal timeout values for a socket object. The *getTimeout* function is an accessor that returns the value of the socket object's internal timeout value. The *setTimeout* and *setTimeout2* functions (which differ only insofar as how you pass the timeout argument) store their argument into the internal data field and they also notify the OS' socket package of the new timeout value.

```
mov( 1, timeValVar.tv_sec );
mov( 500_000, timeValVar.tv_usec );
MyClientSocket.setTimeout( timeValVar );
clientPtr.setTimeout2( 2, 0 );
.
.
.
clientPtr.getTimeout( timeValVar );
```

Note: to ensure consistency with the system, the *getTimeout* function will actually write the internal timeout value to the system. This way, you're ensured that the value that *getTimeout* returns is the timeout value that the operating system will actually use.

```
<object>.getAdrs; @returns( "eax" );
```

This function returns the IP address associated with the object's socket. It returns the IP address in the EAX register in little endian form (not network byte order/big endian form).

```
MyClientSocket.getAdrs();
mov( eax, ipAdrs1 );
clientPtr.getAdrs();
mov( eax, ipAdrs2 );
```

```
<object>.setAdrs( adrs:dword );
```

This function stores the IP address passed as a parameter into the internal address field of the socket object. The *adrs* parameter contains the IP address in little endian form (not network byte order/big endian form). Note that this function is really intended for internal use by the socket classes. This function only stores the IP address

into the internal field. It does not update the IP address of any open socket and it does not change the IP address. The *client\_t.connect* and *server\_t.start* functions provide the mechanism for specifying the IP address of an internet connection. The *setAdrs* function exists so the *start* and *connect* functions can set the IP address in an object-oriented fashion. For that reason, this document is not providing any sample calls to this function.

```
<object>.getPort; @returns( "ax" );
```

This function returns the socket port number associated with the object's socket. It returns the port value in the AX register in little endian form (not network byte order/big endian form).

```
MyClientSocket.getPort();
mov( ax, port1 );
clientPtr.getPort();
mov( ax, port2 );
```

```
<object>.setPort( port:word );
```

This function stores the socket port number passed as a parameter into the internal port field of the socket object. The *port* parameter contains the port value in little endian form (not network byte order/big endian form). Note that this function is really intended for internal use by the socket classes. This function only stores the port number into the object's internal data field. It does not update the port number of any open socket and it does not change the port number in use. The *client\_t.connect* and *server\_t.start* functions provide the mechanism for specifying the port number of an internet connection. The *setPort* function exists so the *start* and *connect* functions can set the port value in an object-oriented fashion. For that reason, this document is not providing any sample calls to this function.

```
<object>.adrsToStr( s:string );
<object>.a_adrsToStr; @returns( "eax" );
```

These functions convert the IP address found in the socket's internal IP address data field to a string of the form "ddd.ddd.ddd.ddd". The *adrsToStr* function stores the string data into the string passed as the argument (raising an exception if that string has insufficient storage); the *a\_adrsToStr* function allocates the storage on the heap and returns a pointer to that string in the EAX register.

```
MyClientSocket.adrsToStr( adrsStr );
clientPtr.a_adrsToStr();
mov( eax, adrsStr2 );
```

## 28.12 Miscellaneous Output

The following socket output routines all assume that you've opened the <object> socket variable via a call to <serverObject>.start or <clientObject>.connect.

```
<object>.write( var buffer:var; count:dword )
```

This method writes the number of bytes specified by the *count* parameter to the socket. The bytes starting at the address of the *buffer* byte are written to the file. No range checking is done on the *buffer*, it is your responsibility to ensure that the buffer contains at least *count* valid data bytes.

Note: Notice that the *buffer* parameter is an untyped reference parameter. Untyped reference parameters have special properties, so be sure to read the chapter on "Passing Parameters to Standard Library Routines" (parmpassing.rtf) if you are not absolutely sure you understand how untyped reference parameters operate.

HLA high-level calling sequence examples:

```

socketPtr.write( buffer, count );

// If bufPtr is a dword object containing the
// address of the buffer whose data you wish to
// write to the socket:

socketPtr.write( val bufPtr, count );

// The following writes the four-byte value of
// the bufPtr variable to the socket (an unusual
// operation):

socketPtr.write( bufPtr, 4 );

```

#### **<object>.putbool( b:boolean );**

This procedure writes the string "true" or "false" to the <object> output socket depending on the value of the *b* parameter.

HLA high-level calling sequence examples:

```

socketPtr.putbool( boolVar );

// If the boolean is in a register (AL):

socketPtr.putbool( al );

```

#### **<object>.newln( );**

This function writes a newline sequence (carriage return/line feed under Windows, linefeed under other operating systems) to the specified socket (<object>).

HLA high-level calling sequence examples:

```

socketPtr.newln();

```

## **28.13 Character, Character Set, and String Output**

The following socket output routines all assume that you've opened the <object> socket variable via a call to <serverObject>.start or <clientObject>.connect.

#### **<object>.putc( c:char )**

Writes the character specified by the *c* parameter to the socket.

HLA high-level calling sequence examples:

```

socketPtr.putc( charVar );

// If the character is in a register (AL):

socketPtr.putc( al );

```

**<object>.putcSize( c:char; width:int32; fill:char )**

Outputs the character *c* to the socket specified by <object> using at least *width* output positions. If the absolute value of *width* is greater than one, then this function writes *fill* characters as padding characters during the output. If *width* is a positive value greater than one, then <object>.putcSize writes *c* left justified in a field of *width* characters; if *width* is a negative value less than one, then <object>.putcSize writes *c* right justified in a field of *width* characters.

HLA high-level calling sequence examples:

```
socketPtr.putcSize( charVar, width, padChar );
```

**<object>.putcset( cst:cset );**

This function writes all the members of the *cst* character set parameter to the specified socket variable.

HLA high-level calling sequence examples:

```
socketPtr.putcset( csVar );
socketPtr.putcset( [ebx] ); // EBX points at the cset.
```

**<object>.puts( s:string );**

This procedure writes the value of the string parameter to the socket.

HLA high-level calling sequence examples:

```
socketPtr.puts( strVar );
socketPtr.puts( ebx ); // EBX holds a string value.
socketPtr.puts( "Hello World" );
```

**<object>.putsSize( s:string; width:int32; fill:char )**

This function writes the *s* string to the socket using at least *width* character positions. If the absolute value of *width* is less than or equal to the length of *s*, then this function behaves exactly like <object>.puts. On the other hand, if the absolute value of *width* is greater than the length of *s*, then <object>.putsSize writes *width* characters to the output file. This procedure emits the *fill* character in the extra print positions. If *width* is positive, then <object>.putsSize right justifies the string in the print field. If *width* is negative, then <object>.putsSize left justifies the string in the print field. Generally, people expect the string to be left justified, so you should ensure that this value is negative to achieve this.

HLA high-level calling sequence examples:

```
socketPtr.putsSize( strVar, width, ' ' );

// For the following, EBX holds the string value,
// ECX contains the width, and AL holds the pad
// character:

socketPtr.putsSize( ebx, ecx, al );

socketPtr.putsSize( "Hello World", 25, padChar );
```

**<object>.putz( z:zstring );**

This procedure writes the value of the zstring parameter to the socket.



HLA high-level calling sequence examples:

```
socketPtr.putz( zstrVar );
socketPtr.putz( ebx ); // EBX holds a zstring value.
socketPtr.putz( "Hello World" );
```

**<object>.putzSize( z:zstring; width:int32; fill:char )**

This function writes the *z* zstring to the socket using at least *width* character positions. If the absolute value of *width* is less than or equal to the length of *z*, then this function behaves exactly like <object>.putz. On the other hand, if the absolute value of *width* is greater than the length of *z*, then <object>.putzSize writes *width* characters to the output file. This procedure emits the *fill* character in the extra print positions. If *width* is positive, then <object>.putzSize right justifies the string in the print field. If *width* is negative, then <object>.putzSize left justifies the string in the print field. Generally, people expect the string to be left justified, so you should ensure that this value is negative to achieve this.

HLA high-level calling sequence examples:

```
socketPtr.putzSize( zstrVar, width, ' ' );

// For the following, EBX holds the string value,
// ECX contains the width, and AL holds the pad
// character:

socketPtr.putzSize( ebx, ecx, al );

socketPtr.putzSize( "Hello World", 25, padChar );
```

## 28.14 Hexadecimal Numeric Output

The following socket output routines all assume that you've opened the <object> socket variable via a call to <serverObject>.start or <clientObject>.connect.

**<object>.putb( b:byte );**

This procedure writes the value of *b* to the socket using exactly two hexadecimal digits (including a leading zero if necessary).

HLA high-level calling sequence examples:

```
socketPtr.putb( byteVar );

// If the character is in a register (AL):

socketPtr.putb( al );
```

**<object>.puth8( b:byte );**

This procedure writes the value of *b* to the socket using one or two hexadecimal digits (the minimum necessary).

HLA high-level calling sequence examples:

```
socketPtr.puth8( byteVar );
```

```
// If the character is in a register (AL):
```

```
socketPtr.puth8( al );
```

**<object>.puth8Size( b:byte; width:dword; fill:char )**

This procedure writes the value of *b* to the socket using a minimum field *width* and a *fill* character. See the discussion of *width* and *fill* in the description of the <object>.putcSize function for more details on their behavior.

HLA high-level calling sequence examples:

```
socketPtr.puth8Size( byteVar, width, padChar );
```

**<object>.putw( w:word );**

This procedure writes the value of *w* to the socket using exactly four hexadecimal digits (including leading zeros if necessary).

HLA high-level calling sequence examples:

```
socketPtr.putw( wordVar );
```

```
// If the word is in a register (AX):
```

```
socketPtr.putw( ax );
```

**<object>.puth16( w:word );**

This procedure writes the value of *w* to the socket using 1-4 hexadecimal digits (the minimum necessary).

HLA high-level calling sequence examples:

```
socketPtr.puth16( wordVar );
```

```
// If the word is in a register (AX):
```

```
socketPtr.puth16( ax );
```

**<object>.puth16Size( w:word; width:dword; fill:char )**

This procedure writes the value of *w* to the socket using a minimum field *width* and a *fill* character. See the discussion of *width* and *fill* in the description of the <object>.putcSize function for more details on their behavior.

HLA high-level calling sequence examples:

```
socketPtr.puth16Size( wordVar, width, padChar );
```

**<object>.putd( dw:dword );**

This procedure writes the value of *d* to the socket using exactly eight hexadecimal digits (including leading zeros if necessary). If the `stdlib` global `underscores` value (see the conversions module for details) contains true, then this function will also print an underscore between the fourth and fifth digits.

HLA high-level calling sequence examples:

```
socketPtr.putd(dwordVar );

// If the dword value is in a register (EAX):

socketPtr.putd( eax );
```

**<object>.puth32( dw:dword );**

This procedure writes the value of *d* to the file using the minimum number of hexadecimal required. If the `stdlib` global `underscores` value (see the conversions module for details) contains true, then this function will also print an underscore between the fourth and fifth digits (if there are at least five digits in the number).

HLA high-level calling sequence examples:

```
socketPtr.puth32( dwordVar );

// If the dword is in a register (EAX):

socketPtr.puth32( eax );
```

**<object>.puth32Size( d:dword; width:dword; fill:char )**

This procedure writes the value of *d* to the socket using a minimum field *width* and a *fill* character. See the discussion of *width* and *fill* in the description of the `<object>.putcSize` function for more details on their behavior.

HLA high-level calling sequence examples:

```
socketPtr.puth32Size( dwordVar, width, ' ' );

// If the dword is in a register (EAX):

socketPtr.puth32Size( eax, width, cl );
```

**<object>.putq( q:qword );**

This procedure writes the value of *q* to the socket using exactly 16 hexadecimal digits (including leading zeros if necessary). If the `stdlib` global `underscores` value (see the conversions module for details) contains true, then this function will also print an underscore between each group of four digits.

HLA high-level calling sequence example:

```
socketPtr.putq( qwordVar );
```

**<object>.puth64( q:qword );**

This procedure writes the value of *q* to the socket using 1-16 hexadecimal digits (the minimum necessary). If the `stdlib` global `underscores` value (see the `conversions` module for details) contains `true`, then this function will also print an underscore between each group of four digits.

HLA high-level calling sequence example:

```
socketPtr.puth64( qwordVar );
```

**<object>.puth64Size( q:qword; width:dword; fill:char )**

This procedure writes the value of *q* to the socket using a minimum field *width* and a *fill* character. See the discussion of *width* and *fill* in the description of the `<object>.putcSize` function for more details on their behavior.

HLA high-level calling sequence example:

```
socketPtr.puth64Size( qwordVar, width, ' ' );
```

**<object>.puttb( tb:tbyte )**

This procedure writes the value of *tb* to the socket using exactly 20 hexadecimal digits (including leading zeros if necessary and an intervening underscores if underscore output is enabled).

HLA high-level calling sequence examples:

```
socketPtr.puttb( tbyteVar );
```

**<object>.puth80( tb:tbyte )**

This procedure writes the value of *tb* to the socket using 1-20 hexadecimal digits (the minimum necessary) and an intervening underscores if underscore output is enabled).

HLA high-level calling sequence examples:

```
socketPtr.puth80( tbyteVar );
```

**<object>.puth80Size( tb:tbyte; width:dword; fill:char )**

This procedure writes the value of *tb* to the socket using a minimum field *width* and a *fill* character. See the discussion of *width* and *fill* in the description of the `<object>.putcSize` function for more details on their behavior.

HLA high-level calling sequence examples:

```
socketPtr.puth80Size( tbyteVar, width, ' ' );
```

**<object>.putl( l:lword )**

This procedure writes the value of *l* to the socket using exactly 32 hexadecimal digits (including leading zeros if necessary and an intervening underscores if underscore output is enabled).

HLA high-level calling sequence examples:

```
socketPtr.putl( lwordVar );
```

**<object>.puth128( l:lword )**

This procedure writes the value of *l* to the socket using 1-32 hexadecimal digits (the minimum necessary) and an intervening underscores if underscore output is enabled).

HLA high-level calling sequence examples:

```
socketPtr.puth128( lwordVar );
```

**<object>.puth128Size( l:lword; width:dword; fill:char )**

This procedure writes the value of *l* to the socket using a minimum field *width* and a *fill* character. See the discussion of *width* and *fill* in the description of the *<object>.putcSize* function for more details on their behavior.

HLA high-level calling sequence examples:

```
socketPtr.puth128Size( tbyteVar, width, ' ' );
```

## 28.15 Signed Integer Numeric Output

The following socket output routines all assume that you've opened the *<object>* socket variable via a call to *<serverObject>.start* or *<clientObject>.connect*.

These routines convert integer values to string format and write that string to the *<object>* socket. The *<object>.putxxxSize* functions contain *width* and *fill* parameters that let you specify the minimum field width when outputting a value.

If the absolute value of *width* is greater than the number of print positions the value requires, then these functions output *width* characters to the socket. If *width* is non-negative, then these functions right-justify the value in the output field; if value is negative, then these functions left-justify the value in the output field.

These functions print the *fill* character as the padding value for the extra print positions.

**xxxSize( value, width, fill );**

Assuming "value" requires five print positions, "width" is eight, and fill is "f" then the xxxSize functions produce the string

f	f	f	V	A	L	U	E
---	---	---	---	---	---	---	---

Assuming "value" requires five print positions, "width" is minus eight, and fill is "f" then the xxxSize functions produce the string

V	A	L	U	E	f	f	f
---	---	---	---	---	---	---	---

**<object>.puti8 ( b:byte );**

This function converts the eight-bit signed integer you pass as a parameter to a string and writes this string to the socket using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
socketPtr.puti8( byteVar );
```

```
// If the character is in a register (AL):
```

```
socketPtr.puti8( al );
```

**<object>.puti8Size ( b:byte; width:int32; fill:char );**

This function writes the eight-bit signed integer value you pass to the specified output socket using the *width* and *fill* values as specified above.

HLA high-level calling sequence examples:

```
socketPtr.puti8Size( byteVar, width, padChar );
```

**<object>.puti16( w:word );**

This function converts the 16-bit signed integer you pass as a parameter to a string and writes this string to the socket using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
socketPtr.puti16( wordVar );
```

```
// If the word is in a register (AX):
```

```
socketPtr.puti16( ax );
```

**<object>.puti16Size( w:word; width:int32; fill:char );**

This function writes the 16-bit signed integer value you pass to the specified socket using the *width* and *fill* values as specified above.

HLA high-level calling sequence examples:

```
socketPtr.puti16Size( wordVar, width, padChar );
```

**<object>.puti32( d:dword );**

This function converts the 32-bit signed integer you pass as a parameter to a string and writes this string to the socket using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
socketPtr.puti32( dwordVar );
```

```
// If the dword is in a register (EAX):
```

```
socketPtr.puti32( eax );
```

**<object>.puti32Size( d:dword; width:int32; fill:char );**

This function writes the 32-bit value you pass as a signed integer to the specified socket using the *width* and *fill* values as specified above.

HLA high-level calling sequence examples:

```
socketPtr.puti32Size( dwordVar, width, ' ' );
```

```
// If the dword is in a register (EAX):
```

```
socketPtr.puti32Size( eax, width, cl );
```

**<object>.puti64( q:qword );**

This function converts the 64-bit signed integer you pass as a parameter to a string and writes this string to the socket using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
socketPtr.puti64( qwordVar );
```

**<object>.puti64Size( q:qword; width:int32; fill:char );**

This function writes the 64-bit value you pass as a signed integer to the specified socket using the *width* and *fill* values as specified above.

HLA high-level calling sequence examples:

```
socketPtr.puti64Size( qwordVar, width, ' ' );
```

**<object>.puti128( l:lword );**

This function converts the 128-bit signed integer you pass as a parameter to a string and writes this string to the socket using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
socketPtr.puti128( lwordVar );
```

```
<object>.puti128Size( l:1word; width:int32; fill:char );
```

This function writes the 128-bit value you pass as a signed integer to the specified socket using the *width* and *fill* values as specified above.

HLA high-level calling sequence examples:

```
socketPtr.puti128Size( lwordVar, width, ' ' );
```

## 28.16 Unsigned Integer Numeric Output

These routines convert unsigned integer values to string format and write that string to the socket. The *<object>.putxxxSize* functions contain *width* and *fill* parameters that let you specify the minimum field width when outputting a value.

If the absolute value of *width* is greater than the number of print positions the value requires, then these functions output *width* characters to the socket. If *width* is non-negative, then these functions right-justify the value in the output field; if value is negative, then these functions left-justify the value in the output field.

These functions print the *fill* character as the padding value for the extra print positions.

```
<object>.putu8 ( b:byte )
```

This function converts the eight-bit unsigned integer you pass as a parameter to a string and writes this string to the socket using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
socketPtr.putu8( byteVar );
```

```
// If the character is in a register (AL):
```

```
socketPtr.putu8( al );
```

```
<object>.putu8size( b:byte; width:int32; fill:char )
```

This function writes the unsigned eight-bit value you pass to the specified socket using the *width* and *fill* values as specified above.

HLA high-level calling sequence examples:

```
socketPtr.putu8Size( byteVar, width, padChar );
```

```
<object>.putu16( w:word )
```

This function converts the 16-bit unsigned integer you pass as a parameter to a string and writes this string to the socket using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
socketPtr.putu16( wordVar );
```

```
// If the word is in a register (AX):
```

```
socketPtr.putu16( ax );
```



**<object>.putu16size( w:word; width:int32; fill:char )**

This function writes the unsigned 16-bit value you pass to the specified socket using the *width* and *fill* values as specified above.

HLA high-level calling sequence examples:

```
socketPtr.putu16Size( wordVar, width, padChar );
```

**<object>.putu32( d:dword )**

This function converts the 32-bit unsigned integer you pass as a parameter to a string and writes this string to the socket using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
socketPtr.putu32( dwordVar );
```

```
// If the dword is in a register (EAX):
```

```
socketPtr.putu32( eax );
```

**<object>.putu32Size( d:dword; width:int32; fill:char )**

This function writes the unsigned 32-bit value you pass to the specified socket using the *width* and *fill* values as specified above.

HLA high-level calling sequence examples:

```
socketPtr.putu32Size( dwordVar, width, ' ' );
```

```
// If the dword is in a register (EAX):
```

```
socketPtr.putu32Size( eax, width, cl );
```

**<object>.putu64( q:qword )**

This function converts the 64-bit unsigned integer you pass as a parameter to a string and writes this string to the socket using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
socketPtr.putu64( qwordVar );
```

**<object>.putu64Size( q:qword; width:int32; fill:char );**

This function writes the unsigned 64-bit value you pass to the specified socket using the *width* and *fill* values as specified above.

HLA high-level calling sequence examples:

```
socketPtr.putu64Size( qwordVar, width, ' ' );
```

**<object>.putu128( l:lword )**

This function converts the 128-bit unsigned integer you pass as a parameter to a string and writes this string to the socket using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
socketPtr.putu128( lwordVar );
```

**<object>.putu128Size( l:lword; width:int32; fill:char );**

This function writes the unsigned 128-bit value you pass to the specified socket using the *width* and *fill* values as specified above.

HLA high-level calling sequence examples:

```
socketPtr.putu128Size( lwordVar, width, ' ' );
```

## 28.17 Floating-Point Numeric Output Using Scientific Notation

The floating point numeric output routines translate the three different binary floating point formats to their string representation and then write this string to the socket that <object> specifies. There are two generic classes of these routines: those that convert their values to exponential/scientific notation and those that convert their string to a decimal form.

The <object>.pute80, <object>.pute64, and <object>.pute32 routines convert their values to a string using scientific notation. These three routines each have two parameters: the value to output and the field width of the result. These routines produce a string with the following format:

s	i	.	f	f	f	f	f	E	±	x
---	---	---	---	---	---	---	---	---	---	---

s is a space for positive values, '-' for negative values  
i represents the integer portion of the mantissa  
ffff represents the fractional portion of the mantissa  
x is one or more base-10 exponent digits.

**<object>.pute32( r:real32; width:uns32 )**

This function writes the 32-bit single precision floating point value passed in *r* to the socket using scientific/exponential notation. This procedure prints the value using *width* print positions in the file. *width* should have a minimum value of five for real numbers in the range 1e-9..1e+9 and a minimum value of six for all other values. Note that 32-bit extended precision floating point values support about 6-7 significant digits. So a *width* value that yields more than seven mantissa digits will produce garbage output in the low order digits of the number.

HLA high-level calling sequence examples:

```

socketPtr.put32( r32Var, width );

// If the real32 value is in an FPU register (ST0):

var
    r32Temp    :real32;
    .
    .
    .
fstp( r32Temp );
socketPtr.put32( r32Temp, 12 );

```

#### **<object>.put64( r:real64; width:uns32 )**

This function writes the 64-bit double precision floating point value passed in *r* to the socket using scientific/exponential notation. This procedure prints the value using *width* print positions in the file. *width* should have a minimum value of five for real numbers in the range 1e-9..1e+9 and a minimum value of six for all other values. Note that 64-bit double precision floating point values support about 15 significant digits. So a *width* value that yields more than 15 mantissa digits will produce garbage output in the low order digits of the number.

HLA high-level calling sequence examples:

```

socketPtr.put64( r64Var, width );

// If the real64 value is in an FPU register (ST0):

var
    r64Temp:real64;
    .
    .
    .
fstp( r64Temp );
socketPtr.put64( r64Temp, 12 );

```

#### **<object>.put80( r:real80; width:uns32 )**

This function writes the 80-bit extended precision floating point value passed in *r* to the socket using scientific/exponential notation. This procedure prints the value using *width* print positions in the file. *width* should have a minimum value of five for real numbers in the range 1e-9..1e+9 and a minimum value of six for all other values. Note that 80-bit extended precision floating point values support about 18 significant digits. So a *width* value that yields more than 18 mantissa digits will produce garbage output in the low order digits of the number.

HLA high-level calling sequence examples:

```

socketPtr.put80( r80Var, width );

// If the real80 value is in an FPU register (ST0):

var
    r80Temp    :real80;
    .
    .
    .
fstp( r80Temp );

```

```
socketPtr.pute80( r80Temp, 12 );
```

## 28.18 Floating-Point Numeric Output Using Decimal Notation

Although scientific (exponential) notation is the most general display format for real numbers, real numbers you display in this format are very difficult to read. Therefore, the HLA socket class module also provides a set of functions that output real values using the decimal representation. Although you cannot (practically) use these decimal output routines for all real values, they are applicable to a wide variety of common numbers you will use in your programs.

These functions come in two varieties. The first variety requires four parameters: the real value to convert, the width of the converted value, the number of digit positions to the right of the decimal point, and a padding character. The second variety only requires the first three parameters and assumes the padding character is a space. These functions write their values using the following string format:

s	i	i	i	.	f	f	f	f	f	f
---	---	---	---	---	---	---	---	---	---	---

s is a space for positive values, '-' for negative values  
 i represents the integer portion of the mantissa  
 ffff represents the fractional portion of the mantissa

```
<object>.putr32( r:real32; width:uns32; decpts:uns32; fill:char )
```

This procedure writes a 32-bit single precision floating point value to the socket as a string. The string consumes exactly *width* characters in the output file. If the numeric output, using the specified number of positions to the right of the decimal point, is sufficiently small that the string representation would be less than *width* characters, then this procedure uses the *fill* value as the padding character to fill the output with *width* characters.

HLA high-level calling sequence examples:

```
socketPtr.putr32( r32Var, width, decpts, fill );
socketPtr.putr32( r32Var, 10, 2, '*' );

// If the real32 value is in an FPU register (ST0):

var
    r32Temp    :real32;
    .
    .
    .
fstp( r32Temp );
socketPtr.putr32( r32Temp, 12, 2, al );
```

```
<object>.putr64( r:real64; width:uns32; decpts:uns32; fill:char )
```

This procedure writes a 64-bit double precision floating point value to <object> socket as a string. The string consumes exactly *width* characters in the output. If the numeric output, using the specified number of positions to the right of the decimal point, is sufficiently small that the string representation would be less than *width* characters, then this procedure uses the value of *fill* as the padding character to fill the output with *width* characters.

HLA high-level calling sequence examples:

```

socketPtr.putr64( r64Var, width, decpts, fill );
socketPtr.putr64( r64Var, 10, 2, '*' );

// If the real64 value is in an FPU register (ST0):

var
    r64Temp    :real64;
    .
    .
    .
fstp( r64Temp );
socketPtr.putr64( r64Temp, 12, 2, al );

```

**<object>.putr80( r:real80; width:uns32; decpts:uns32; fill:char )**

This procedure writes an 80-bit extended precision floating point value to the socket as a string. The string consumes exactly *width* characters in the output. If the numeric output, using the specified number of positions to the right of the decimal point, is sufficiently small that the string representation would be less than *width* characters, then this procedure uses the value of *fill* as the padding character to fill the output with *width* characters.

HLA high-level calling sequence examples:

```

socketPtr.putr80( r80Var, width, decpts, fill );
socketPtr.putr80( r80Var, 10, 2, '*' );

// If the real80 value is in an FPU register (ST0):

var
    r80Temp    :real80;
    .
    .
    .
fstp( r80Temp );
socketPtr.putr80( r80Temp, 12, 2, al );

```

## 28.19 Generic File Output

**<object>.put( parameter\_list )**

*<object>.put* is a macro that automatically invokes an appropriate *<object>* output routine based on the type of the parameter(s) you pass it. This is a very convenient output routine and is probably the socket class formatted output call you will use most often in your programs. Keep in mind that this macro is not a single function call; instead, HLA translates this macro into a sequence of calls to procedures like *<object>.puti32*, *<object>.puts*, etc.

*<object>.put* is a macro that provides a flexible syntax for outputting data to the socket. This macro allows a variable number of parameters. For each parameter present in the list, *<object>.put* will call the appropriate routine to emit that data, according to the type of the parameter. Parameters may be constants, registers, or memory locations. You must separate each macro parameter with a comma.

Here is an example of a typical invocation of an *<object>.put* :

```

<object>.put( "I=", i, " j=", j, nl );

```

The above is roughly equivalent to

```
<object>.puts( "I=" );
<object>.puti32( i );
<object>.puts( " j=" );
<object>.puti32( j );
<object>.newln();
```

This assumes, of course, that *i* and *j* are *int32* variables.

The `<object>.put` macro also lets you specify the minimum field width for each parameter you specify. To print a value using a minimum field width, follow the object you wish to print with a colon and the value of the minimum field width. The previous example, using field widths, could look like the following:

```
<object>.put( "I=", i:2, " j=", j:5, nl );
```

Although this example used the literal decimal constants two and five for the field widths, keep in mind that register values and memory value (integers, anyway) are perfectly legal here.

For floating point numbers you wish to display in decimal form, you can specify both the minimum field width and the number of digits to print to the right of the decimal point by using the following syntax:

```
<object>.put( "Real value is ", f:10:3, nl );
```

The `<object>.put` macro can handle all the basic primitive types, including boolean, unsigned (8, 16, 32, 64, 128), signed (8, 16, 32, 64, 128), character, character set, real (32, 64, 80), string, and hexadecimal (byte, word, dword, qword, tbyte, lword).

If you specify a class variable (object) and that class defines a *toString* method, the `<object>.put` macro will call the associated *toString* method and output that string to the socket. Note that the *toString* method must dynamically allocate storage for the string by calling *str.alloc*. This is because `<object>.put` will call *str.free* on the string once it outputs the string.

There is a known "design flaw" in the `<object>.put` macro. You cannot use it to print HLA intermediate variables (i.e., non-local VAR objects). The problem is that HLA's syntax for non-local accesses takes the form "reg32:varname" and `<object>.put` cannot determine if you want to print *reg32* using *varname* print positions versus simply printing the non-local *varname* object. If you want to display non-local variables you must copy the non-local object into a register, a static variable, or a local variable prior to using `<object>.put` to print it. Of course, there is no problem using the other `<object>.putXXXX` functions to display non-local VAR objects, so you can use those as well.

**Important(!)**, don't forget that method calls (e.g., the routines that `<object>.put` translates into) modify the values in the ESI and EDI registers. Therefore, it never makes any sense to attempt to print the values of ESI and EDI within the parameter list. All you will wind up doing is printing the address of the file variable (ESI) or the address of its virtual method table (EDI). If you need to write these two values to a file, move them to another register or a memory location first.

## 28.20 Generic File Input

The following socket input routines behave just like their standard input and file input counterparts (unless otherwise noted). Because of the nature of sockets, it is not possible to provide an "end-of-file" function that tests whether you're currently at the end of file on an input stream. End of file is determined by a timeout (set by the *setTimeout* and *setTimeout2* functions). Whenever a timeout occurs while the program is waiting for an input from a socket, the system translates that timeout into an *ex.EndOfFile* exception. Therefore, you should really surround all socket input requests with a **try..endtry** sequence that handles the *ex.EndOfFile* exception.

```
<object>.read( var buffer:var; count:dword )
```

This will probably be the most commonly-called input function in a typical socket-based application. This function reads *count* bytes from the socket and stores them into memory starting with the first byte of the *buffer* variable. This routine does not do any range checking. It is your responsibility to ensure that *buffer* is large enough to hold the data read.

Note: Notice that the *buffer* parameter is an untyped reference parameter. Untyped reference parameters have special properties, so be sure to read the chapter on "Passing Parameters to Standard Library Routines" (parmpassing.rtf) if you are not absolutely sure you understand how untyped reference parameters operate.

HLA high-level calling sequence examples:

```
socketPtr.read( buffer, count );
socketPtr.read( [eax], 1024 );
```

### **<object>.readLn;**

This function reads and discards all characters up to and including the newline sequence in the socket stream.

HLA high-level calling sequence examples:

```
socketPtr.readLn();
```

## **28.21 Character and String Input**

The following functions read character data from a socket. Note that HLA's socket class module does not provide the ability to read character set data directly from the user. However, you can always read a string and then convert that string to a character set using the appropriate function in the *cset* module.

### **<object>.getc; @returns( "al" );**

This function reads a single character from the socket and returns that character in the AL register.

HLA high-level calling sequence examples:

```
socketPtr.getc();
mov( al, charVar );
```

### **<object>.gets( s:string );**

This function reads a sequence of characters from the socket through to the next end of line sequence and stores these characters (without the end of line sequence) into the string variable you pass as a parameter. Before calling this routine, you must allocate sufficient storage for the string. If *<object>.gets* attempts to read a larger string than the string's *MaxLen* value, *<object>.gets* raises a string overflow exception.

Note that this function does not store the end of line sequence into the string, though it does consume the end of line sequence. The next character a file class function will read from the socket will be the first character of the following line.

If the incoming socket data is a newline sequence, then *<object>.gets* consumes the end of line and stores the empty string into the *s* parameter.

HLA high-level calling sequence examples:

```
socketPtr.gets( inputStr );
socketPtr.gets( eax ); // EAX contains string value
```

### **<object>.a\_gets; @returns( "eax" );**

Like *<object>.gets*, this function also reads a string from the socket. However, rather than storing the string data into a string you supply, this function allocates storage for the string on the heap and returns a pointer to this string in the EAX register. You code should call *str.free* to release this storage when you're done with the string data.

The *<object>.a\_gets* function imposes a line length limit of 4,096 characters. If this is a problem, you should modify the source code for this function to raise the limit. This function raises an exception if you attempt to read a line longer than this internal limit.

HLA high-level calling sequence examples:

```
socketPtr.a_gets();
mov( eax, inputStr );
```

## 28.22 Signed Integer Input

The functions in this group read numeric values from the socket using a signed decimal integer format. These functions read the string data, translate it to numeric form, and return that numeric data in an appropriate location.

```
<object>.geti8; @returns( "al" );
```

This function reads a signed eight-bit decimal integer in the range -128..+127 from the socket. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by an optional minus sign and a string of one or more decimal digits. The number must end with a valid delimiter character or the end of file (i.e., timeout). This function allows underscores in the interior of the number. The *<object>.geti8* function raises an appropriate exception if the input violates any of these rules or the value is outside the range -128..+127. This function returns the binary form of the integer in the AL register.

HLA high-level calling sequence examples:

```
socketPtr.geti8();
mov( al, i8Var );
```

```
<object>.geti16; @returns( "ax" );
```

This function reads a signed 16-bit decimal integer in the range -32768..+32767 from the socket. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by an optional minus sign and a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file (i.e., timeout). This function allows underscores in the interior of the number. The *<object>.geti16* function raises an appropriate exception if the input violates any of these rules or the value is outside the range -32768..+32767. This function returns the binary form of the integer in the AX register.

HLA high-level calling sequence examples:

```
socketPtr.geti16();
mov( ax, i16Var );
```

```
<object>.geti32; @returns( "eax" );
```

This function reads a signed 32-bit decimal integer in the (approximate) range  $\pm 2$  Billion from the socket. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by an optional minus sign and a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file (i.e., timeout). This function allows underscores in the interior of the number. The *<object>.geti32* function raises an appropriate exception if the input violates any of these rules or the value is outside the range plus or minus two billion. This function returns the binary form of the integer in the EAX register.

HLA high-level calling sequence examples:

```
socketPtr.geti32();
```



```
mov( eax, i32Var );
```

```
<object>.geti64; @returns( "edx:eax" );
```

This function reads a signed 64-bit decimal integer from the socket. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by an optional minus sign and a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file (i.e., timeout). This function allows underscores in the interior of the number. The *<object>.geti64* function raises an appropriate exception if the input violates any of these rules or the value is outside the range of a 64-bit signed integer. This function returns the 64-bit result in the EDX:EAX register pair (it returns the H.O. dword in EDX and the L.O. dword in EAX).

HLA high-level calling sequence examples:

```
socketPtr.geti64();
mov( edx, (type dword i64Var[4]) );
mov( eax, (type dword i64Var[0]) );
```

```
<object>.geti128( var l:lword );
```

This function reads a signed 128-bit decimal integer from the socket. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by an optional minus sign and a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file (i.e., timeout). This function allows underscores in the interior of the number. The *<object>.geti128* function raises an appropriate exception if the input violates any of these rules or the value is outside the range of a 128-bit signed integer. This function stores the 128-bit result into the lword you pass as a reference parameter.

HLA high-level calling sequence examples:

```
socketPtr.geti128( lwordVar );
```

## 28.23 Unsigned Integer Input

The functions in this group read numeric values from the socket using an unsigned decimal integer format. These functions read the string data, translate it to numeric form, and return that numeric data in an appropriate location.

```
<object>.getu8; @returns( "al" );
```

This function reads an unsigned eight-bit decimal integer in the range 0..+255 from the socket. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file (i.e., timeout). This function allows underscores in the interior of the number. The *<object>.getu8* function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..255. This function returns the binary form of the integer in the AL register.

HLA high-level calling sequence examples:

```
socketPtr.getu8();
mov( al, u8Var );
```

```
<object>.getu16; @returns( "ax" );
```

This function reads an unsigned 16-bit decimal integer in the range 0..+65535 from the socket. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file (i.e., timeout). This function allows underscores in the interior of the number. The *<object>.getu16* function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..65535. This function returns the binary form of the integer in the AX register.

HLA high-level calling sequence examples:

```
socketPtr.getu16();
mov( ax, u16Var );
```

```
<object>.getu32; @returns( "eax" );
```

This function reads an unsigned 32-bit decimal integer in the range 0..+4,294,967,295 from the socket. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file (i.e., timeout). This function allows underscores in the interior of the number. The *<object>.getu32* function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..4,294,967,295. This function returns the binary form of the integer in the EAX register.

HLA high-level calling sequence examples:

```
socketPtr.getu32();
mov( eax, u32Var );
```

```
<object>.getu64; @returns( "edx:eax" );
```

This function reads an unsigned 64-bit decimal integer from the socket. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file (i.e., timeout). This function allows underscores in the interior of the number. The *<object>.getu64* function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..2<sup>64</sup>-1. This function returns the binary form of the integer in EDX:EAX register pair (EDX contains the H.O. dword, EAX holds the L.O. dword).

HLA high-level calling sequence examples:

```
socketPtr.getu32();
mov( eax, (type dword u64Var) );
mov( edx, (type dword u64Var[4]) );
```

```
<object>.getu128( var l:lword );
```

This function reads an unsigned 128-bit decimal integer from the socket. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file (i.e., timeout). This function allows underscores in the interior of the number. The *<object>.getu64* function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..2<sup>128</sup>-1. This function returns the binary form of the integer in the lword parameter you pass by reference.

HLA high-level calling sequence examples:

```
socketPtr.getu128( u128Var );
```

## 28.24 Hexadecimal Input

```
<object>.geth8; @returns( "al" );
```

This function reads an eight-bit hexadecimal integer in the range 0..\$FF from the socket. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "\$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character or the end of the file (i.e., timeout). This function allows underscores in the interior of the number. The *<object>.geth8* function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..\$FF. This function returns the binary form of the value in the AL register.

HLA high-level calling sequence examples:

```
socketPtr.geth8();
mov( al, h8Var );
```

```
<object>.geth16; @returns( "ax" );
```

This function reads a 16-bit hexadecimal integer in the range 0..\$FFFF from the socket. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "\$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character or the end of the file (i.e., timeout). This function allows underscores in the interior of the number. The *<object>.geth16* function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..\$FFFF. This function returns the binary form of the value in the AX register.

HLA high-level calling sequence examples:

```
socketPtr.geth16();
mov( ax, h16Var );
```

```
<object>.geth32; @returns( "eax" );
```

This function reads a 32-bit hexadecimal integer in the range 0..\$FFFF\_FFFF from the socket. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "\$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character or the end of the file (i.e., timeout). This function allows underscores in the interior of the number. The *<object>.geth32* function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..\$FFFF\_FFFF. This function returns the binary form of the value in the EAX register.

HLA high-level calling sequence examples:

```
socketPtr.geth32();
mov( eax, h32Var );
```

```
<object>.geth64; @returns( "edx:eax" );
```

This function reads a 64-bit hexadecimal integer in the range 0..\$FFFF\_FFFF\_FFFF\_FFFF from the socket. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "\$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character or the end of the file (i.e., timeout). This function allows underscores in the interior of the number. The *<object>.geth64* function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..\$FFFF\_FFFF\_FFFF\_FFFF. This function returns the 64-bit result in the EDX:EAX register pair (EDX contains the H.O. dword, EAX contains the L.O. dword).

HLA high-level calling sequence examples:

```
socketPtr.geth64();
mov( edx, (type dword h64Var[4]) );
mov( eax, (type dword h64Var[0]) );
```

```
<object>.geth128( var l:1word );
```

This function reads a 128-bit hexadecimal integer in the range 0..\$FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF from the socket. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "\$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character or the end of the file (i.e., timeout). This function allows underscores in the interior of the number. The *<object>.getq* function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..\$FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF. This function stores the 128-bit result into the variable you pass as a reference parameter.

HLA high-level calling sequence examples:

```
socketPtr.geth128( lwordVar );
```

## 28.25 Floating-Point Input

```
<object>.getf; @returns( "st0" );
```

This function reads an 80-bit floating point value in either decimal or scientific from the socket and leaves the result sitting on the FPU stack. The number may begin with any number of delimiter characters (see the *conv.setDelimiter* and *conv.getDelimiter* functions for details on the delimiter characters) followed by an optional minus sign and a sequence of characters that represent a floating point value. The number must end with a valid delimiter character or the end of the file (i.e., timeout). This function allows underscores in the interior of the number. This function raises an appropriate exception if an error occurs.

HLA high-level calling sequence examples:

```
socketPtr.getf();
fstp( fpVar );
```

## 28.26 Generic File Input

```
<object>.get( List_of_items_to_read );
```

This is a macro that allows you to specify a list of variable names as parameters. The `<object>.get` macro reads an input value for each item in the list and stores the resulting value in each corresponding variable. This macro determines the type of each variable that you pass it and emits a call to the appropriate `<object>.getxxx` function to read the actual value. As an example, consider the following call to `<object>.get`:

```
socketPtr.get( i32, charVar, u16, strVar );
```

The macro invocation above expands into the following:

```
push( eax );
socketPtr.geti32( i32 );
socketPtr.getc();
mov( al, charVar );
socketPtr.geti16();
mov( ax, u16 );
socketPtr.gets( strVar );
pop( eax );
```

Notice that `<object>.get` preserves the value in the EAX register even though various `<object>.getxxx` functions use this register. Note that `<object>.get` automatically handles the case where you specify EAX as an input variable and writes the value to `[esp]` so that it properly modifies EAX upon completion of the macro expansion.

Note that `<object>.get` only supports eight-, sixteen-, and thirty-two bit integer input. If you need to read 64-bit or 128-bit values, you must use the appropriate `<object>.getx64` or `<object>.getx128` function to achieve this.



## 29 The Standard Error Module (stderr.hhf)

This unit contains routines that write data to the standard error device. This is usually the console device, although the user may redirect the standard error to a file from the command line.

Note: be sure to read the chapter on "Passing Parameters to Standard Library Routines" (parmpassing.rtf) before reading this chapter.

**Note about stack diagrams:** this documentation includes stack diagrams for those functions that pass parameters on the stack. To conserve space, this documentation does not include a stack diagram for any function that does not pass data on the stack (that is, only a return address appears on the stack).

**A Note About Thread Safety:** Because the standard error device is a single resource, you will get inconsistent results if multiple threads attempt to write to the standard error device simultaneously. The HLA standard library stderr module does not attempt to synchronize thread access to the standard error device. If you are going to be writing to the standard error from multiple threads, it is your responsibility to ensure that the threads use properly synchronized access to this resource.

**A Note About the FPU:** The Standard Library code makes occasional use of the FPU, particularly when converting between real and string formats and when computing certain mathematical functions. You should exercise caution when using MMX instructions in a program that makes use of the Standard Library. In particular, you should ensure that you are always in FPU mode (by executing an EMMS instruction) after you are finished using MMX instructions. Better yet, you should avoid the MMX instruction set altogether and use the improved SSE instruction set that accomplishes the same tasks (and doesn't disturb the FPU).

### 29.1 Conversion Format Control

The standard error functions that convert numeric values to hexadecimal, unsigned decimal, and signed decimal output provide the ability to inject underscores between groups of three (decimal) or four (hexadecimal) digits to make it easier to read large numbers. You enable and disable underscore output using the `conv.setUnderscores` and `conv.getUnderscores` functions. Please refer to their documentation in the `conv.rtf` file for more details.

When converting numeric values to string form for output, the standard error routines call the conversion functions found in the `conv` (conversions) module. For detailed information on the actual conversions, please consult the `conv.rtf` document.

### 29.2 File I/O Routines and the Standard Error Handle

The standard error routines are basically a thin layer over the top of the fileio routines (see the fileio documentation for a complete description of those routines). Indeed, if you obtain the standard error handle, you can write data to the standard error device by passing this handle to a fileio function. Because the fileio module provides a slightly richer set of routines, there are a few instances where you might want to do this. You might also want to write a generic output function that expects a file handle and then pass it the standard error device file handle so that the function writes its output to the console (or other standard error device) rather than to some file. In any case, just be aware that it is perfectly reasonable to call fileio functions to write data to the standard error device.

```
stderr.handle; @returns( "eax" );
```

This routine returns the Linux/Windows handle for the Standard Error Device in the EAX register. You may use this handle with the file I/O routines to write data to the standard error device.

### 29.3 Standard Error Routines

The output routines in the stderr module are very similar to the file output routines in the stderr module. In general, these routines require (at least) one parameter: the value to write to the standard error device. Some functions contain additional parameters that provide formatting information.

## 29.4 Miscellaneous Output Routines

```
stderr.write( var buffer:var; count:uint32 );
```

This procedure writes the number of bytes specified by the count variable to the standard error device. The bytes starting at the address of the buffer variable are written to standard err. No range checking is done on the buffer, it is your responsibility to ensure that the buffer contains at least count valid data bytes. Because the buffer parameter is passed by untyped reference, a high-level style call to this function will take the address of whatever object you supply as the buffer parameter. *This includes pointer variables* (which is probably not what you want to do). Use the VAL keyword in a high-level style call if you want to use the value of a pointer variable rather than the address of that pointer variable (see the examples that follow).

HLA high-level calling sequence examples:

```
stderr.write( buffer, count );

// If "bufPtr" is dword containing the address of the buffer, then
// use the following code:

stderr.write( val bufPtr, bufferSize );

// If you actually want to write out the four bytes held by
// bufPtr (an unusual thing to do), you would use the
// following code:

stderr.write( bufPtr, 4 );
```

HLA low-level calling sequence examples:

```
// Assumes buffer is a static object at a fixed
// address in memory:

pushd( &buffer );
push( count );
call stderr.write;

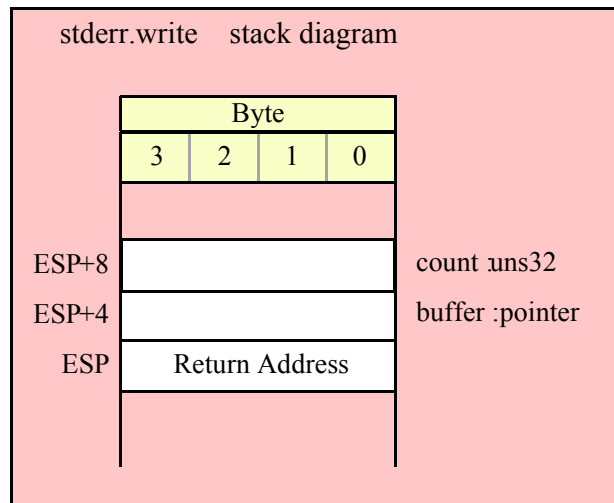
// If a 32-bit register is available and buffer
// isn't at a fixed, static, address:

lea( eax, buffer );
push( eax );
push( count );
call stderr.write;

// If a 32-bit register is not available and buffer
// isn't at a fixed, static, address:

sub( 4, esp );
push( eax );
lea( eax, buffer );
mov( eax, [esp+4] );
pop( eax );
push( count );
call stderr.write;
```



**stderr.newln()**

This function writes a newline sequence (e.g., carriage return/line feed under Windows or line feed under Linux) to the standard error output.

HLA high-level calling sequence examples:

```
stderr.newln();
```

HLA low-level calling sequence examples:

```
call stderr.newln;
```

## 29.5 Boolean Output

**stderr.putbool( b:boolean );**

This procedure writes the string "true" or "false" to the standard error depending on the value of the b parameter.

HLA high-level calling sequence examples:

```
stderr.putbool( boolVar );
```

```
// If the boolean is in a register (AL):
```

```
stderr.putbool( al );
```

HLA low-level calling sequence examples:

```
// If "boolVar" is not one of the last three
// bytes on a page of memory, you can do this:
```

```

push( (type dword boolVar ) );
call stderr.putbool;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( boolVar , eax ); // Assume EAX is available
push( eax );
call stderr.putbool;

// If no register is available, do something
// like the following code:

sub( 4, esp );
push( eax );
movzx( boolVar , eax );
mov( eax, [esp+4] );
pop( eax );
call stderr.putbool;

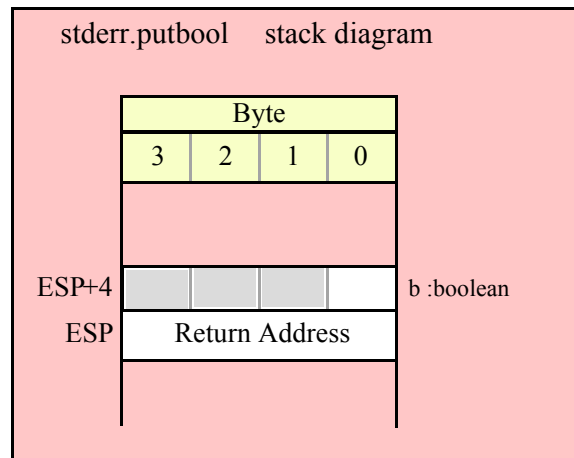
// If the boolean value is in al, bl, cl, or dl
// then you can use code like the following:

push( eax ); // Assume boolVar is in AL
call stderr.putbool;

// If the Boolean value is in ah, bh, ch, or dh
// you'll have to use code like the following:

xchg( al, ah ); // Assume boolVar is in AH
push( eax );    // It's now in AL
xchg( al, ah ); // Restore al/ah
call stderr.putbool;

```



## 29.6 Character, String, and Character Set Output Routines

```
stderr.putc( c:char );
```

Writes the character specified by the *c* parameter to the standard error device.

HLA high-level calling sequence examples:

```

stderr.putc( charVar );

// If the character is in a register (AL):

stderr.putc( al );

```

HLA low-level calling sequence examples:

```

// If "charVar" is not one of the last three
// bytes on a page of memory, you can do this:

push( (type dword charVar) );
call stderr.putc;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( charVar, eax ); // Assume EAX is available
push( eax );
call stderr.putc;

// If no register is available, do something
// like the following code:

sub( 4, esp );
push( eax );
movzx( charVar, eax );
mov( eax, [esp+4] );
pop( eax );
call stderr.putc;

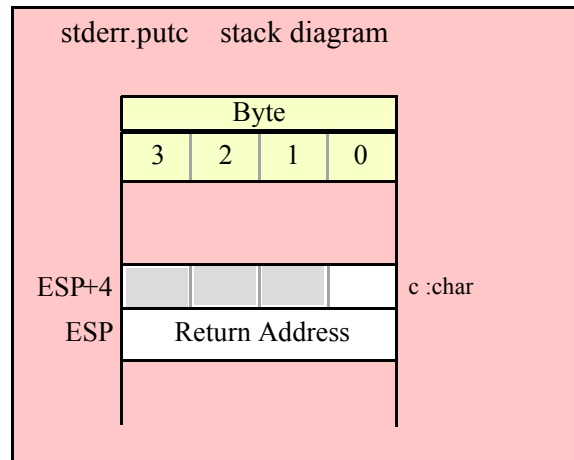
// If the character value is in al, bl, cl, or dl
// then you can use code like the following:

push( eax ); // Assume charVar is in AL
call stderr.putc;

// If the character value is in ah, bh, ch, or dh
// you'll have to use code like the following:

xchg( al, ah ); // Assume charVar is in AH
push( eax );    // It's now in AL
xchg( al, ah ); // Restore al/ah
call stderr.putc;

```



**stderr.putcSize( c:char; width:int32; fill:char )**

Outputs the character *c* to the standard error using at least *width* output positions. If the absolute value of *width* is greater than one, then this function writes *fill* characters as padding characters during the output. If *width* is a positive value greater than one, then *stderr.putcSize* writes *c* left justified in a field of *width* characters; if *width* is a negative value less than one, then *stderr.putcSize* writes *c* right justified in a field of *width* characters.

HLA high-level calling sequence examples:

```
stderr.putcSize( charVar, width, padChar );
```

HLA low-level calling sequence examples:

```
// If "charVar" and "padChar" are not one of the last three
// bytes on a page of memory, you can do this:

push( (type dword charVar) );
push( width );
push( (type dword padChar) );
call stderr.putcSize;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( charVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call stderr.putcSize;

// If no registers are available, do something
// like the following code:

push( eax );
movzx( charVar, eax );
push( eax );
push( width );
```

```

movzx( padChar, eax );
push( eax );
call stderr.putcSize;
pop( eax );

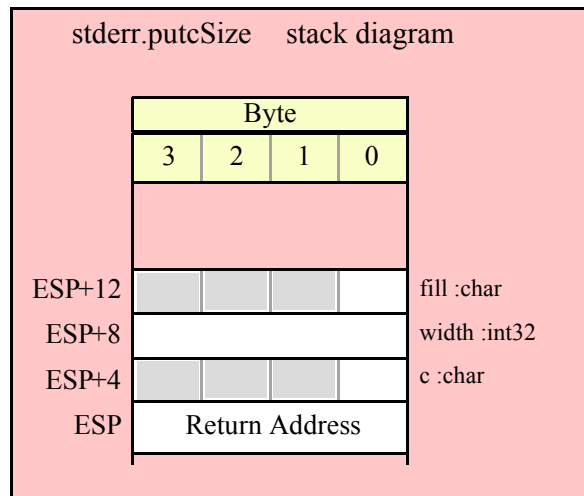
// If "charVar" or "padChar" are in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:

push( eax );    // Assume charVar is in AL
push( width );
push( ebx );    // Assume padChar is in BL
call stderr.putcSize;

// Do the following if the characters are
// in AH, BH, CH, or DH:

xchg( al, ah );    // Assume charVar is in AH
xchg( bl, bh );    // Assume padChar is in BH
push( eax );
push( width );
push( ebx );
xchg( al, ah );
xchg( bl, bh );
call stderr.putcSize;

```



**stderr.putcset( cst:cset );**

This function writes all the members of the `cst` character set parameter to the standard error device.

HLA high-level calling sequence examples:

```

stderr.putcset( csVar );
stderr.putcset( [ebx] ); // EBX points at the cset.

```

HLA low-level calling sequence examples:

```

push( (type dword csVar[12]) ); // Push H.O. dword first
push( (type dword csVar[8]) );
push( (type dword csVar[4]) );

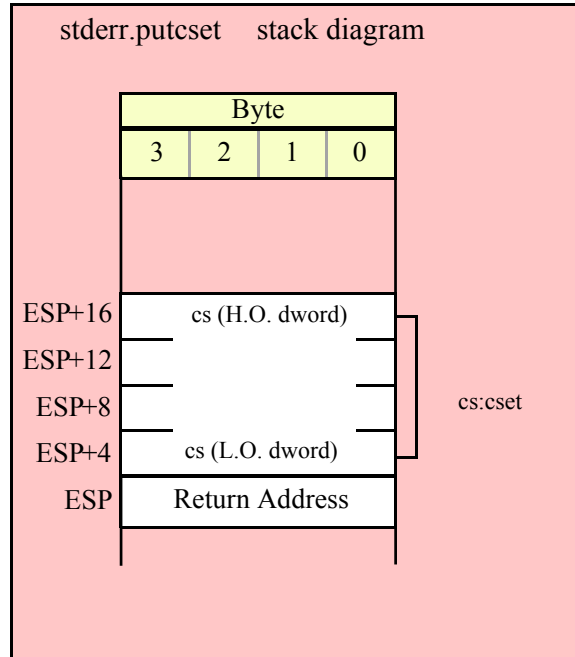
```

```

push( (type dword csVar) );      // Push L.O. dword last
call stderr.putcset;

push( (type dword [ebx+12]) );   // Push H.O. dword first
push( (type dword [ebx+8]) );
push( (type dword [ebx+4]) );
push( (type dword [ebx]) );      // Push L.O. dword last
call stderr.putcset;

```



**stderr.puts( s:string );**

This procedure writes the value of the string parameter to the standard error. Remember, string values are actually 4-byte pointers to the string's character data.

HLA high-level calling sequence examples:

```

stderr.puts( strVar );
stderr.puts( ebx ); // EBX holds a string value.
stderr.puts( "Hello World" );

```

HLA low-level calling sequence examples:

// For string variables:

```

push( strVar );
call stderr.puts;

```

// For string values held in registers:

```

push( ebx ); // Assume EBX holds the string value
call stderr.puts;

```

```

// For string literals, assuming a 32-bit register
// is available:

```

```

lea( eax, "Hello World" ); // Assume EAX is available.

```

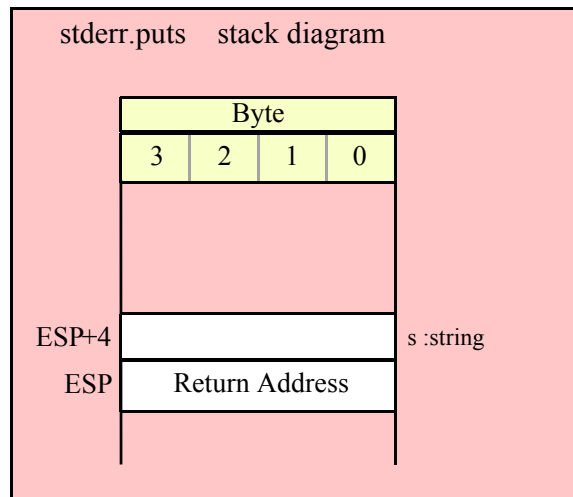
```

push( eax );
call stderr.puts;

// If a 32-bit register is not available:

readonly
    literalString :string := "Hello World";
    .
    .
    .
push( literalString );
call stderr.puts;

```



```
stderr.putsSize( s:string; width:int32; fill:char );
```

This function writes the `s` string to the standard error using at least `width` character positions. If the absolute value of `width` is less than or equal to the length of `s`, then this function behaves exactly like `stderr.puts`. On the other hand, if the absolute value of `width` is greater than the length of `s`, then `stderr.putsSize` writes `width` characters to the standard error. This procedure emits the fill character in the extra print positions. If `width` is positive, then `stderr.putsSize` right justifies the string in the print field. If `width` is negative, then `stderr.putsSize` left justifies the string in the print field. Generally, people expect the string to be left justified, so you should ensure that this value is negative to achieve this.

HLA high-level calling sequence examples:

```

stderr.putsSize( strVar, width, ' ' );

// For the following, EBX holds the string value,
// ECX contains the width, and AL holds the pad
// character:

stderr.putsSize( ebx, ecx, al );

stderr.putsSize( "Hello World", 25, padChar );

```

HLA low-level calling sequence examples:

```

// For string variables:

push( strVar );
push( width );

```

```

pushd( ' ' );
call stderr.putsSize;

// For string values held in registers:

push( ebx ); // Assume EBX holds the string value
push( ecx ); // Assume ECX holds the width
push( eax ); // Assume AL holds the fill character
call stderr.putsSize;

// For string literals, assuming a 32-bit register
// is available:

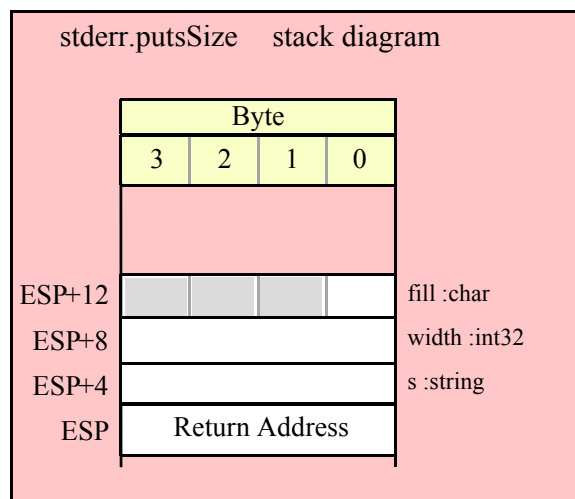
lea( eax, "Hello World" ); // Assume EAX is available.
push( eax );
pushd( 25 );
movzx( padChar, eax );
push( eax );
call stderr.putsSize;

// If a 32-bit register is not available:

readonly
literalString :string := "Hello World";

// Note: element zero is the actual pad character.
// The other elements are just padding.
padChar :char[4] := [ '.', #0, #0, #0 ];
.
.
.
push( literalString );
pushd( 25 );
push( (type dword padChar) );
call stderr.putsSize;

```



## 29.7 Hexadecimal Output Routines

These routines convert numeric data to hexadecimal string form (using the hexadecimal conversion routines found in the conv module) and write the resulting string to the standard error device.



**stderr.putb( b:byte )**

This procedure writes the value of *b* to the standard error using exactly two hexadecimal digits (including a leading zero if necessary).

HLA high-level calling sequence examples:

```
stderr.putb( byteVar );

// If the character is in a register (AL):

stderr.putb( al );
```

HLA low-level calling sequence examples:

```
// If "byteVar " is not one of the last three
// bytes on a page of memory, you can do this:

push( (type dword byteVar ) );
call stderr.putb;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( byteVar , eax ); // Assume EAX is available
push( eax );
call stderr.putb;

// If no register is available, do something
// like the following code:

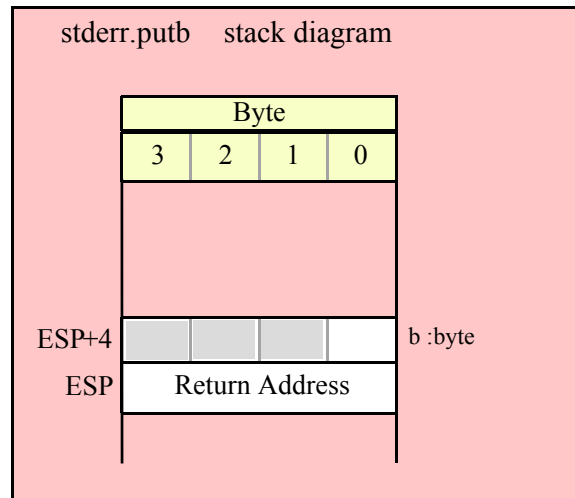
sub( 4, esp );
push( eax );
movzx( byteVar , eax );
mov( eax, [esp+4] );
pop( eax );
call stderr.putb;

// If the character value is in al, bl, cl, or dl
// then you can use code like the following:

push( eax ); // Assume byteVar is in AL
call stderr.putb;

// If the byte value is in ah, bh, ch, or dh
// you'll have to use code like the following:

xchg( al, ah ); // Assume byteVar is in AH
push( eax );    // It's now in AL
xchg( al, ah ); // Restore al/ah
call stderr.putb;
```



```
stderr.putb8( b:byte );
```

This procedure writes the value of `b` to the standard error using the minimum necessary number of hexadecimal digits.

HLA high-level calling sequence examples:

```
stderr.putb8( byteVar );
```

```
// If the character is in a register (AL):
```

```
stderr.putb8( al );
```

HLA low-level calling sequence examples:

```
// If "byteVar " is not one of the last three
// bytes on a page of memory, you can do this:
```

```
push( (type dword byteVar ) );
call stderr.putb8;
```

```
// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:
```

```
movzx( byteVar , eax ); // Assume EAX is available
push( eax );
call stderr.putb8;
```

```
// If no register is available, do something
// like the following code:
```

```
sub( 4, esp );
push( eax );
movzx( byteVar , eax );
mov( eax, [esp+4] );
```

```

pop( eax );
call stderr.puth8;

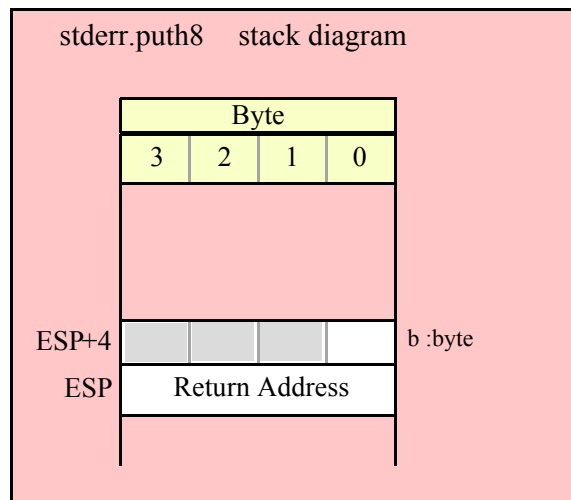
// If the character value is in al, bl, cl, or dl
// then you can use code like the following:

push( eax ); // Assume byteVar is in AL
call stderr.puth8;

// If the byte value is in ah, bh, ch, or dh
// you'll have to use code like the following:

xchg( al, ah ); // Assume byteVar is in AH
push( eax );    // It's now in AL
xchg( al, ah ); // Restore al/ah
call stderr.puth8;

```



**stderr.puth8Size( b:byte; size:dword; fill:char )**

The `stderr.puth8Size` function writes an 8-bit hexadecimal value to the standard error allowing you specify a minimum field width and a fill character.

HLA high-level calling sequence examples:

```
stderr.puth8Size( byteVar, width, padChar );
```

HLA low-level calling sequence examples:

```

// If "byteVar" and "padChar" are not one of the last three
// bytes on a page of memory, you can do this:

push( (type dword byteVar) );
push( width );
push( (type dword padChar) );
call stderr.puth8Size;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

```

```

movzx( byteVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call stderr.puth8Size;

// If no registers are available, do something
// like the following code:

push( eax );
movzx( byteVar, eax );
push( eax );
push( width );
movzx( padChar, eax );
push( eax );
call stderr.puth8Size;
pop( eax );

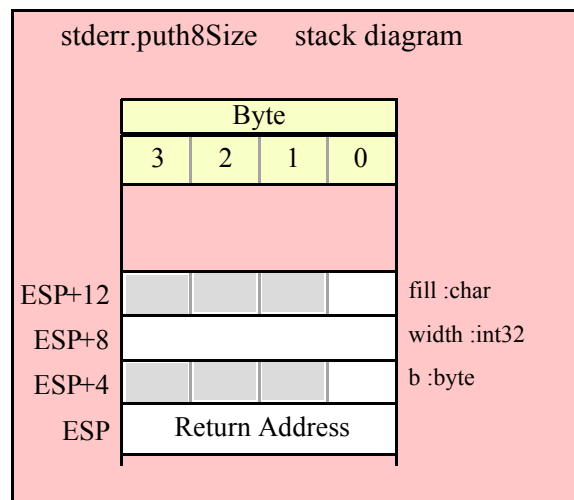
// If "byteVar" or "padChar" are in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:

push( eax ); // Assume byteVar is in AL
push( width );
push( ebx ); // Assume padChar is in BL
call stderr.puth8Size;

// Do the following if the characters are
// in AH, BH, CH, or DH:

xchg( al, ah ); // Assume byteVar is in AH
xchg( bl, bh ); // Assume padChar is in BH
push( eax );
push( width );
push( ebx );
xchg( al, ah );
xchg( bl, bh );
call stderr.puth8Size;

```



**stderr.putw( w:word )**

This procedure writes the value of w to the standard error device using exactly four hexadecimal digits (including leading zeros if necessary).

HLA high-level calling sequence examples:

```
stderr.putw( wordVar );

// If the word is in a register (AX):

stderr.putw( ax );
```

HLA low-level calling sequence examples:

```
// If "wordVar " is not one of the last three
// bytes on a page of memory, you can do this:

push( (type dword wordVar) );
call stderr.putw;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

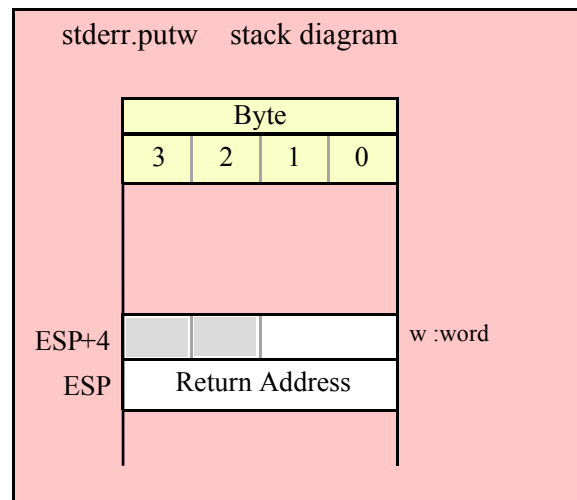
movzx( wordVar, eax ); // Assume EAX is available
push( eax );
call stderr.putw;

// If no register is available, do something
// like the following code:

push( eax ):
movzx( wordVar, eax );
push( eax );
call stderr.putw;
pop( eax );

// If the word value is in a 16-bit register
// then you can use code like the following:

push( eax ); // Assume wordVar is in AX
call stderr.putw;
```



```
stderr.puth16( w:word )
```

This procedure writes the value of w to the standard err using the minimum necessary number of hexadecimal digits.

HLA high-level calling sequence examples:

```
stderr.puth16( wordVar );
```

```
// If the word is in a register (AX):
```

```
stderr.puth16( ax );
```

HLA low-level calling sequence examples:

```
// If "wordVar " is not one of the last three
// bytes on a page of memory, you can do this:
```

```
push( (type dword wordVar) );
call stderr.puth16;
```

```
// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:
```

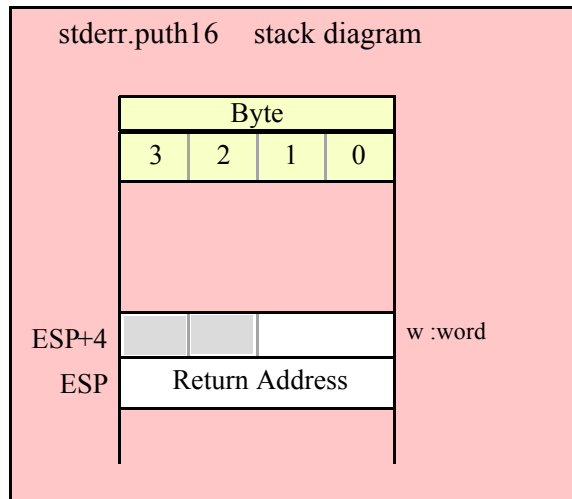
```
movzx( wordVar, eax ); // Assume EAX is available
push( eax );
call stderr.puth16;
```

```
// If no register is available, do something
// like the following code:
```

```
push( eax );
movzx( wordVar, eax );
push( eax );
call stderr.puth16;
pop( eax );
```

```
// If the word value is in a 16-bit register
// then you can use code like the following:

push( eax ); // Assume wordVar is in AX
call stderr.puth16;
```



**stderr.puth16Size( w:word; size:dword; fill:char )**

The `stderr.puth16Size` function writes a 16-bit hexadecimal value to the standard error allowing you specify a minimum field width and a fill character.

HLA high-level calling sequence examples:

```
stderr.puth16Size( wordVar, width, padChar );
```

HLA low-level calling sequence examples:

```
// If "wordVar" and "padChar" are not one of the last three
// bytes on a page of memory, you can do this:

push( (type dword wordVar) );
push( width );
push( (type dword padChar) );
call stderr.puth16Size;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

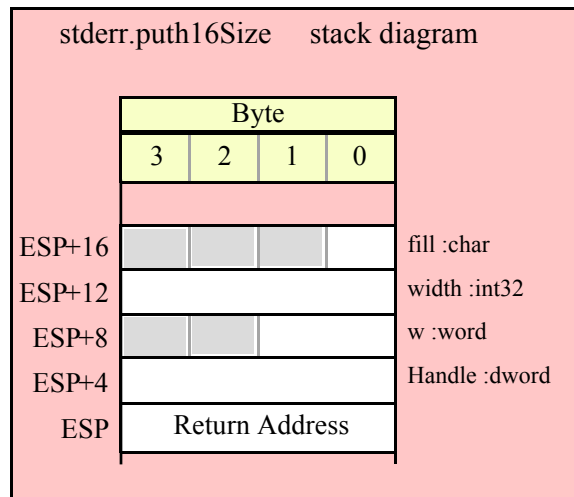
movzx( wordVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call stderr.puth16Size;

// If no registers are available, do something
// like the following code:
```

```
push( eax );
movzx( wordVar, eax );
push( eax );
push( width );
movzx( padChar, eax );
push( eax );
call stderr.puth16Size;
pop( eax );

// If "wordVar" is in a 16-bit register
// and "padChar" is in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:

push( eax );    // Assume wordVar is in AX
push( width );
push( ebx );    // Assume padChar is in BL
call stderr.puth16Size;
```



```
stderr.putd( d:dword )
```

This procedure writes the value of `d` to the standard err using exactly eight hexadecimal digits (including leading zeros if necessary), if underscore output is not enabled. This routine will emit nine characters (eight digits plus an underscore) if underscore output is enabled.

HLA high-level calling sequence examples:

```
stderr.putd( dwordVar );

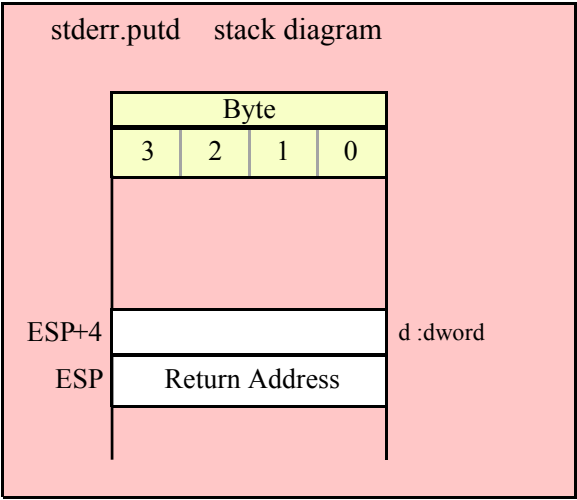
// If the dword value is in a register (EAX):
stderr.putd( eax );
```

HLA low-level calling sequence examples:

```
push( dwordVar );
call stderr.putd;
```



```
push( eax );  
call stderr.putd;
```



```
stderr.puth32( d:dword );
```

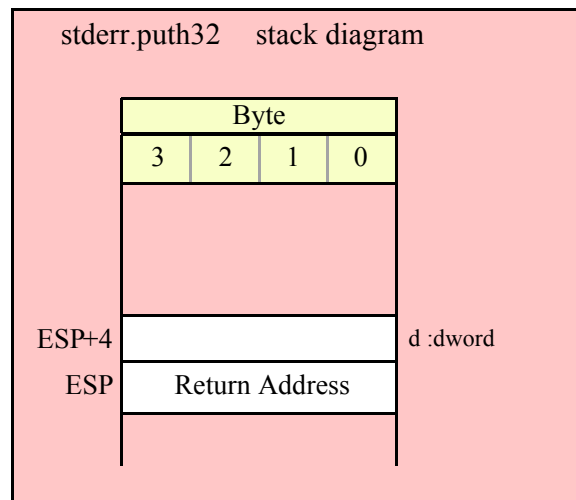
This procedure writes the value of d to the standard error using the minimum number of hexadecimal digits necessary. If underscore output is enabled (see conv.setUnderscores and conv.getUnderscores) then this function will emit an underscore between groups of four hexadecimal digits, starting from the least significant digit.

HLA high-level calling sequence examples:

```
stderr.puth32( dwordVar );  
  
// If the dword is in a register (EAX):  
  
stderr.puth32( eax );
```

HLA low-level calling sequence examples:

```
push( dwordVar );  
call stderr.puth32;  
  
push( eax );  
call stderr.puth32;
```



### **stderr.puth32Size( d:dword; size:dword; fill:char )**

The stderr.puth32Size function outputs d as a hexadecimal string (including underscores, if enabled) and it allows you specify a minimum field width and a fill character.

HLA high-level calling sequence examples:

```
stderr.puth32Size( dwordVar, width, ' ' );
```

```
// If the dword is in a register (EAX):
```

```
stderr.puth32Size( eax, width, cl );
```

HLA low-level calling sequence examples:

```
push( dwordVar );
push( width );
pushd( ' ' );
call stderr.puth32Size;
```

```
push( eax );
push( width );
push( ecx ); // fill char is in CL
call stderr.puth32Size;
```

```
// Assume fill char is in CH
```

```
push( eax );
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call stderr.puth32Size;
```

```
// Alternate method of the above
```

```
push( eax );
push( width );
```

```

sub( 4, esp );
mov( ch, [esp] );
call stderr.puth32Size;

// If the fill char is a variable and
// a register is available, try this code:

push( eax );
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call stderr.puth32Size;

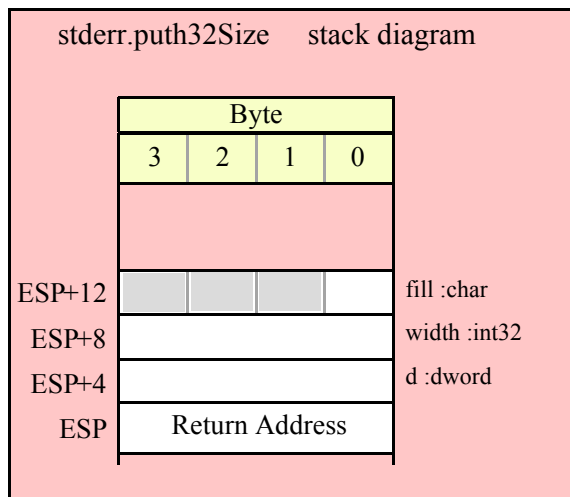
// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( eax );
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call stderr.puth32Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( eax );
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call stderr.puth32Size;

```



```
stderr.putq( q:qword );
```

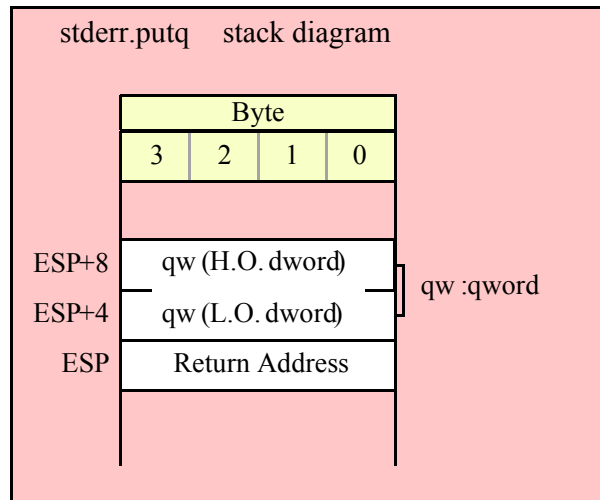
This procedure writes the value of *q* to the standard error device using exactly sixteen hexadecimal digits (including leading zeros if necessary and an intervening underscores if underscore output is enabled).

HLA high-level calling sequence examples:

```
stderr.putq( qwordVar );
```

HLA low-level calling sequence examples:

```
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
call stderr.putq;
```



```
stderr.puth64( q:qword );
```

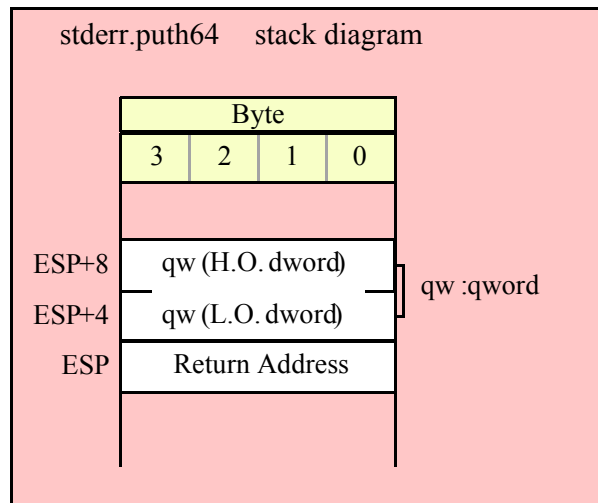
This procedure writes the value of *q* to the standard error using the minimum necessary number of hexadecimal digits (including intervening underscores if underscore output is enabled).

HLA high-level calling sequence examples:

```
stderr.puth64( qwordVar );
```

HLA low-level calling sequence examples:

```
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
call stderr.puth64;
```



```
stderr.puth64Size( q:qword; size:dword; fill:char );
```

The `stderr.putqSize` function lets you specify a minimum field width and a fill character. The `stderr.putq` routine uses a minimum size of two and a fill character of '0'. Note that if underscore output is enabled, this routine will emit 19 characters (16 digits plus three underscores).

HLA high-level calling sequence examples:

```
stderr.puth64Size( qwordVar, width, ' ' );
```

HLA low-level calling sequence examples:

```
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
pushd( ' ' );
call stderr.puth64Size;
```

```
push( edx ); // Assume 64-bit value in edx:eax
push( eax );
push( width );
push( ecx ); // fill char is in CL
call stderr.puth64Size;
```

```
// Assume fill char is in CH
```

```
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call stderr.puth64Size;
```

```
// Alternate method of the above
```

```
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
```

```

push( width );
sub( 4, esp );
mov( ch, [esp] );
call stderr.puth64Size;

// If the fill char is a variable and
// a register is available, try this code:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));     // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call stderr.puth64Size;

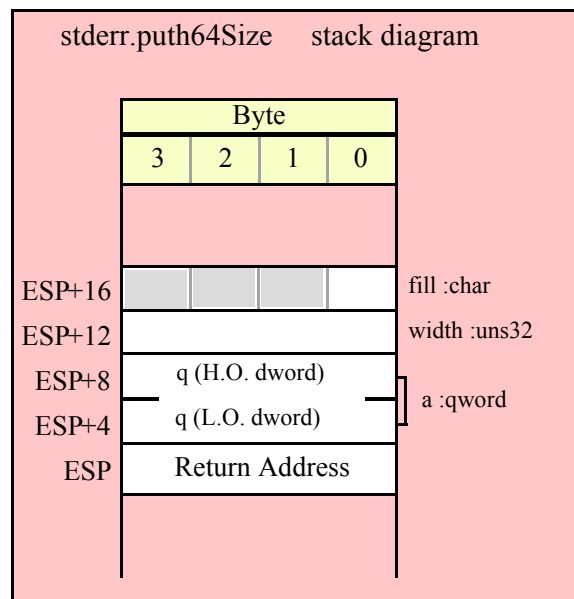
// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));     // L.O. dword last
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call stderr.puth64Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));     // L.O. dword last
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call stderr.puth64Size;

```



**stderr.puttb( tb:tbyte );**

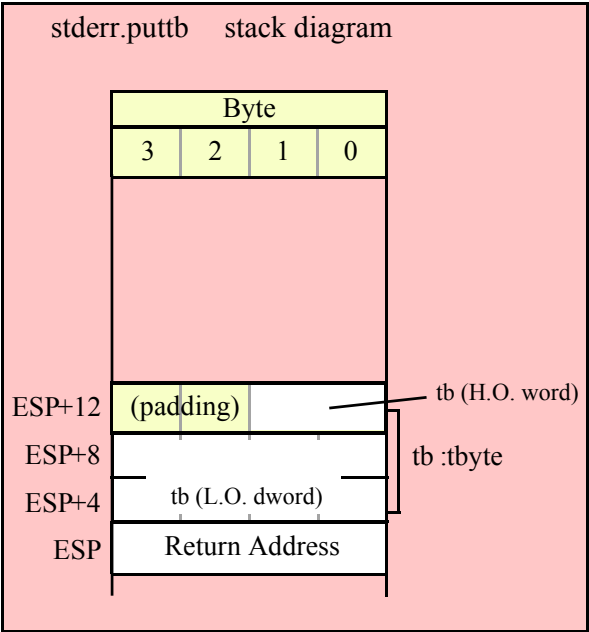
This procedure writes the value of tb to the standard err using exactly 20 hexadecimal digits (including leading zeros if necessary and an intervening underscores if underscore output is enabled).

HLA high-level calling sequence examples:

```
stderr.puttb( tbyteVar );
```

HLA low-level calling sequence examples:

```
pushw( 0 ); // Push push a 0 pad word
push( (type word tbyteVar[8])); // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar)); // L.O. dword last
call stderr.puttb;
```



**stderr.puth80( tb:tbyte );**

This procedure writes the value of tb to the standard error using the minimum necessary number of hexadecimal digits (including intervening underscores if underscore output is enabled).

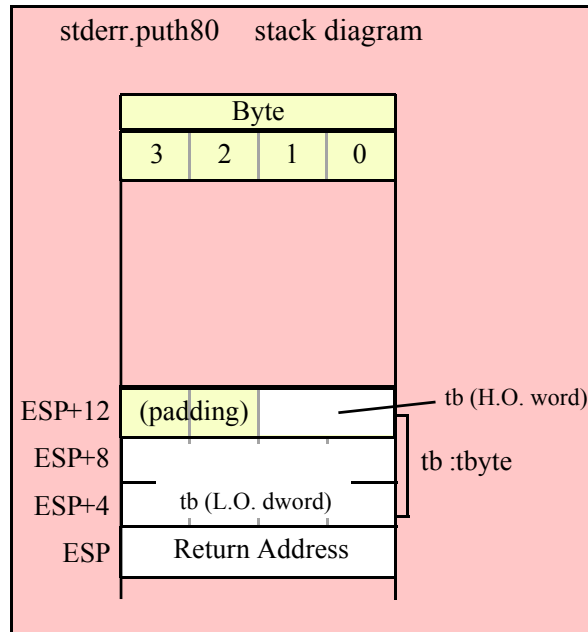
HLA high-level calling sequence examples:

```
stderr.puth80( tbyteVar );
```

HLA low-level calling sequence examples:

```
pushw( 0 ); // Push push a 0 pad word
push( (type word tbyteVar[8])); // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
```

```
push( (type dword tbyteVar));    // L.O. dword last
call stderr.puth80;
```



```
stderr.puth80Size( tb:tbyte; size:dword; fill:char );
```

The `stderr.puth80Size` function lets you specify a minimum field width and a fill character. It writes the `tbyte` value `tb` as a hexadecimal string to the standard error device using the provided minimum size and fill character.

HLA high-level calling sequence examples:

```
stderr.puth80Size( tbyteVar, width, ' ' );
```

HLA low-level calling sequence examples:

```
pushw( 0 );                // Push push a 0 pad word
push( (type word tbyteVar[8])); // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));    // L.O. dword last
push( width );
pushd( ' ' );
call stderr.puth80Size;
```

```
// Assume fill char is in CH
```

```
pushw( 0 );                // Push push a 0 pad word
push( (type word tbyteVar[8])); // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));    // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call stderr.puth80Size;
```



```

// Alternate method of the above

pushw( 0 );                // Push push a 0 pad word
push( (type word tbyteVar[8])); // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));    // L.O. dword last
push( width );
sub( 4, esp );
mov( ch, [esp] );
call stderr.puth80Size;

// If the fill char is a variable and
// a register is available, try this code:

pushw( 0 );                // Push push a 0 pad word
push( (type word tbyteVar[8])); // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));    // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call stderr.puth80Size;

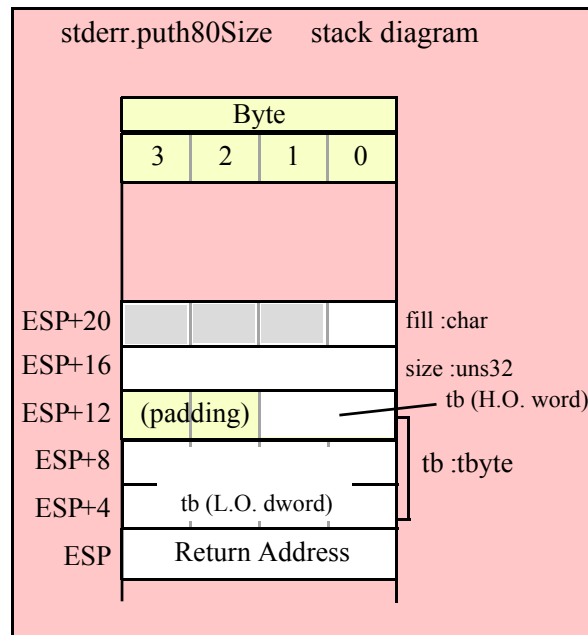
// If the fill char is a variable and
// no register is available, here's one
// possibility:

pushw( 0 );                // Push push a 0 pad word
push( (type word tbyteVar[8])); // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));    // L.O. dword last
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call stderr.puth80Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

pushw( 0 );                // Push push a 0 pad word
push( (type word tbyteVar[8])); // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));    // L.O. dword last
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call stderr.puth80Size;

```



```
stderr.putl( l:ldword );
```

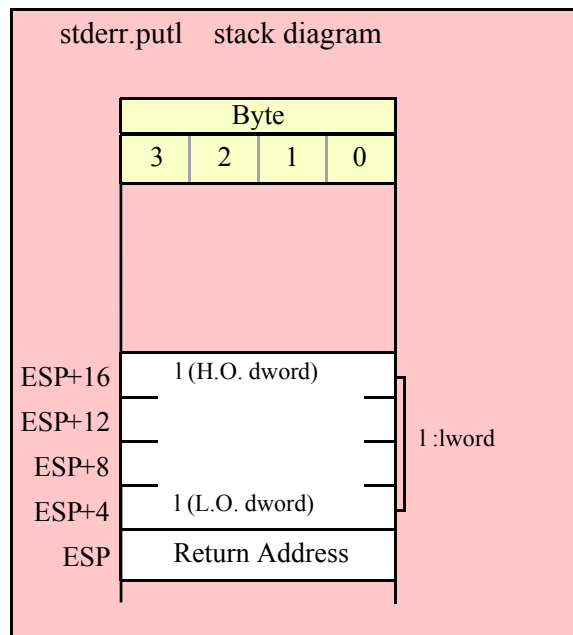
This procedure writes the value of `l` to the standard error using exactly 32 hexadecimal digits (including leading zeros if necessary and an intervening underscores if underscore output is enabled).

HLA high-level calling sequence examples:

```
stderr.putl( lwordVar );
```

HLA low-level calling sequence examples:

```
push( (type dword lwordVar[12]) ); // H.O. dword first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
call stderr.putl;
```



```
stderr.puth128( l:1word );
```

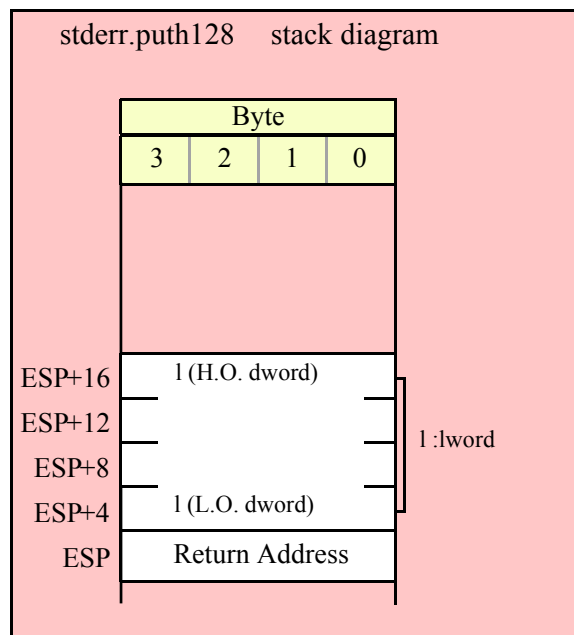
This procedure writes the value of `l` to the standard error using the minimum necessary number of hexadecimal digits (including intervening underscores if underscore output is enabled).

HLA high-level calling sequence examples:

```
stderr.puth128( lwordVar );
```

HLA low-level calling sequence examples:

```
push( (type dword lwordVar[12]) ); // H.O. dword first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
call stderr.puth128;
```



```
stderr.puth128Size( 1:lword; size:dword; fill:char );
```

The `std::err::put128Size` function writes an `lword` value to the standard error and it lets you specify a minimum field width and a fill character.

HLA high-level calling sequence examples:

```
stderr.puth128Size( tbyteVar, width, ' ' );
```

HLA low-level calling sequence examples:

```
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar)); // L.O. dword last
push( width );
pushd( ' ' );
call stderr.puth128Size;
```

```
// Assume fill char is in CH
```

```
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar)); // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call stderr.puth128Size;
```

```
// Alternate method of the above
```

```

push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar)); // L.O. dword last
push( width );
sub( 4, esp );
mov( ch, [esp] );
call stderr.puth128Size;

// If the fill char is a variable and
// a register is available, try this code:

push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar)); // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call stderr.puth128Size;

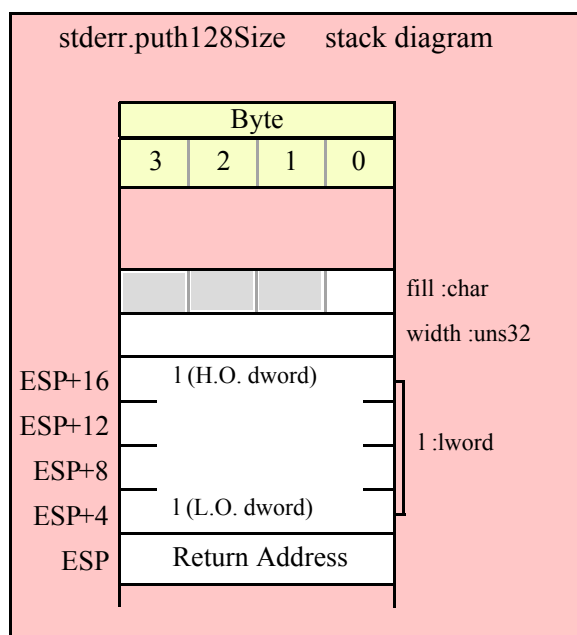
// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar)); // L.O. dword last
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call stderr.puth128Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar)); // L.O. dword last
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call stderr.puth128Size;

```



## 29.8 Signed Integer Output Routines

These routines convert signed integer values to string format and write that string to the standard error device. The `stderr.putxxxSize` functions contain width and fill parameters that let you specify the minimum field width when outputting a value.

If the absolute value of width is greater than the number of print positions the value requires, then these functions output width characters to the standard error device. If width is non-negative, then these functions right-justify the value in the output field; if value is negative, then these functions left-justify the value in the output field.

These functions print the fill character as the padding value for the extra print positions.

Note that unlike floating point values, these functions do not print a space in front of the value if it is non-negative.

```
xxxSize( value, width, fill );
```

Assuming “value” requires five print positions, “width” is eight, and fill is “f” then the `xxxSize` functions produce the string

f	f	f	V	A	L	U	E
---	---	---	---	---	---	---	---

Assuming “value” requires five print positions, “width” is minus eight, and fill is “f” then the `xxxSize` functions produce the string

V	A	L	U	E	f	f	f
---	---	---	---	---	---	---	---

```
stderr.puti8 ( b:byte );
```

This function converts the eight-bit signed integer you pass as a parameter to a string and writes this string to the standard error using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
stderr.puti8( byteVar );
```

```
// If the character is in a register (AL):
```

```
stderr.puti8( al );
```

HLA low-level calling sequence examples:

```
// If "byteVar " is not one of the last three
// bytes on a page of memory, you can do this:
```

```
push( (type dword byteVar ) );
call stderr.puti8;
```

```
// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:
```

```
movzx( byteVar , eax ); // Assume EAX is available
push( eax );
call stderr.puti8;
```

```
// If no register is available, do something
// like the following code:
```

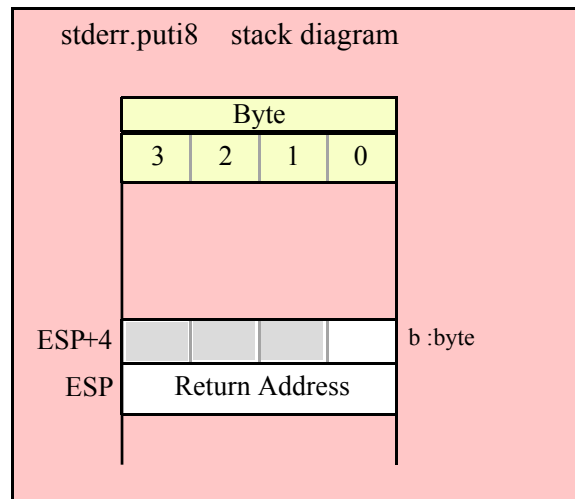
```
push( eax );
movzx( byteVar , eax );
push( eax );
call stderr.puti8;
pop( eax );
```

```
// If the character value is in al, bl, cl, or dl
// then you can use code like the following:
```

```
push( eax ); // Assume byteVar is in AL
call stderr.puti8;
```

```
// If the byte value is in ah, bh, ch, or dh
// you'll have to use code like the following:
```

```
xchg( al, ah ); // Assume byteVar is in AH
push( eax );    // It's now in AL
xchg( al, ah ); // Restore al/ah
call stderr.puti8;
```



**stderr.puti8Size ( b:byte; width:int32; fill:char )**

This function writes the eight-bit signed integer value you pass to the standard error using the width and fill values as specified above.

HLA high-level calling sequence examples:

```
stderr.puti8Size( byteVar, width, padChar );
```

HLA low-level calling sequence examples:

```
// If "byteVar" and "padChar" are not one of the last three
// bytes on a page of memory, you can do this:
```

```
push( (type dword byteVar) );
push( width );
push( (type dword padChar) );
call stderr.puti8Size;
```

```
// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:
```

```
movzx( byteVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call stderr.puti8Size;
```

```
// If no registers are available, do something
// like the following code:
```

```
push( eax );
movzx( byteVar, eax );
push( eax );
push( width );
movzx( padChar, eax );
push( eax );
```



```

call stderr.puti8Size;
pop( eax );

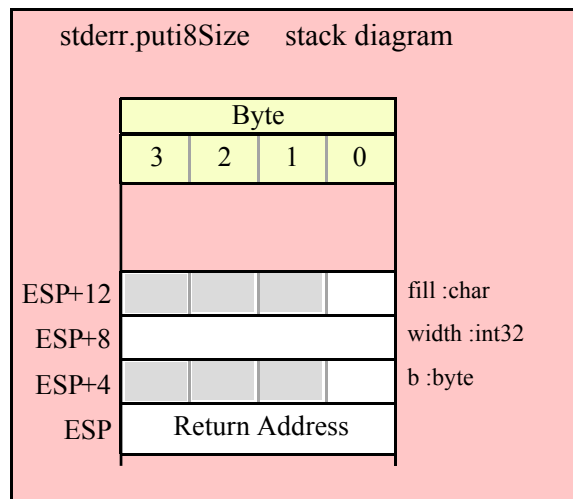
// If "byteVar" or "padChar" are in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:

push( eax );    // Assume byteVar is in AL
push( width );
push( ebx );    // Assume padChar is in BL
call stderr.puti8Size;

// Do the following if the characters are
// in AH, BH, CH, or DH:

xchg( al, ah );    // Assume byteVar is in AH
xchg( bl, bh );    // Assume padChar is in BH
push( eax );
push( width );
push( ebx );
xchg( al, ah );
xchg( bl, bh );
call stderr.puti8Size;

```



### **stderr.puti16( w:word );**

This function converts the 16-bit signed integer you pass as a parameter to a string and writes this string to the standard error device using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```

stderr.puti16( wordVar );

// If the word is in a register (AX):

stderr.puti16( ax );

```

HLA low-level calling sequence examples:

```

// If "wordVar " is not one of the last three
// bytes on a page of memory, you can do this:

push( (type dword wordVar) );
call stderr.puti16;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( wordVar, eax ); // Assume EAX is available
push( eax );
call stderr.puti16;

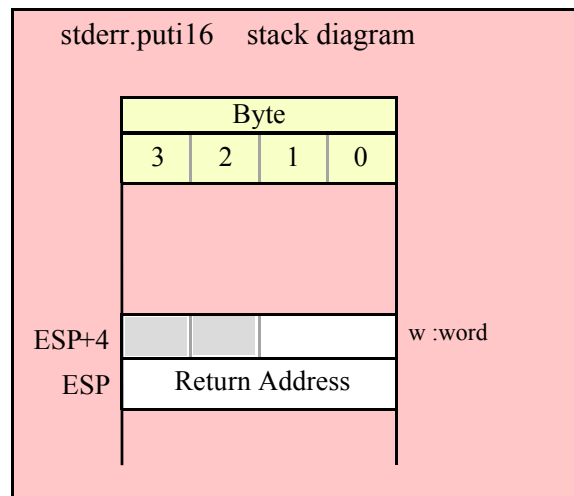
// If no register is available, do something
// like the following code:

push( eax );
movzx( wordVar, eax );
push( eax );
call stderr.puti16;
pop( eax );

// If the word value is in a 16-bit register
// then you can use code like the following:

push( eax ); // Assume wordVar is in AX
call stderr.puti16;

```



```
stderr.puti16Size( w:word; width:int32; fill:char );
```

This function writes the 16-bit signed integer value you pass to the standard error using the width and fill values as specified above.

HLA high-level calling sequence examples:

```
stderr.puti16Size( wordVar, width, padChar );
```

HLA low-level calling sequence examples:

```
// If "wordVar" and "padChar" are not one of the last three
// bytes on a page of memory, you can do this:

push( (type dword wordVar) );
push( width );
push( (type dword padChar) );
call stderr.puti16Size;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

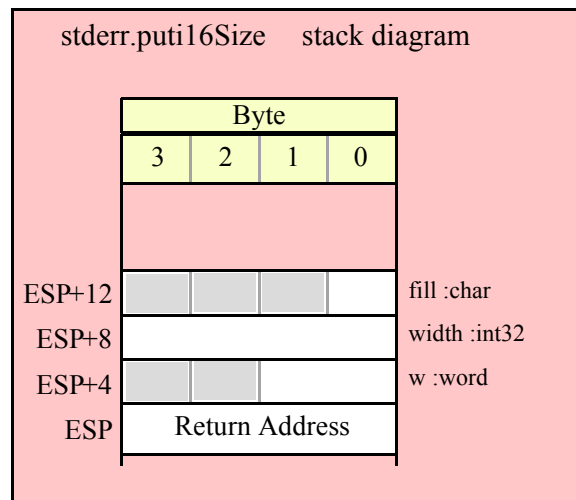
movzx( wordVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call stderr.puti16Size;

// If no registers are available, do something
// like the following code:

push( eax );
movzx( wordVar, eax );
push( eax );
push( width );
movzx( padChar, eax );
push( eax );
call stderr.puti16Size;
pop( eax );

// If "wordVar" is in a 16-bit register
// and "padChar" is in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:

push( eax ); // Assume wordVar is in AX
push( width );
push( ebx ); // Assume padChar is in BL
call stderr.puti16Size;
```



```
stderr.puti32( d:dword );
```

This function converts the 32-bit signed integer you pass as a parameter to a string and writes this string to the standard err using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
stderr.puti32( dwordVar );
```

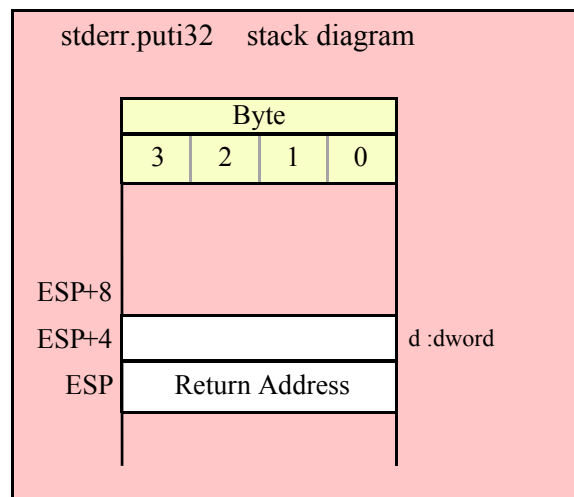
```
// If the dword is in a register (EAX):
```

```
stderr.puti32( eax );
```

HLA low-level calling sequence examples:

```
push( dwordVar );
call stderr.puti32;
```

```
push( eax );
call stderr.puti32;
```



```
stderr.puti32Size( d:dword; width:int32; fill:char );
```

This function writes the 32-bit value you pass as a signed integer to the standard error device using the width and fill values as specified above.

HLA high-level calling sequence examples:

```
stderr.putu32Size( dwordVar, width, ' ' );

// If the dword is in a register (EAX):

stderr.putu32Size( eax, width, cl );
```

HLA low-level calling sequence examples:

```
push( dwordVar );
push( width );
pushd( ' ' );
call stderr.putu32Size;

push( eax );
push( width );
push( ecx ); // fill char is in CL
call stderr.putu32Size;

// Assume fill char is in CH

push( eax );
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call stderr.putu32Size;

// Alternate method of the above

push( eax );
push( width );
sub( 4, esp );
mov( ch, [esp] );
call stderr.putu32Size;

// If the fill char is a variable and
// a register is available, try this code:

push( eax );
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call stderr.putu32Size;

// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( eax );
```

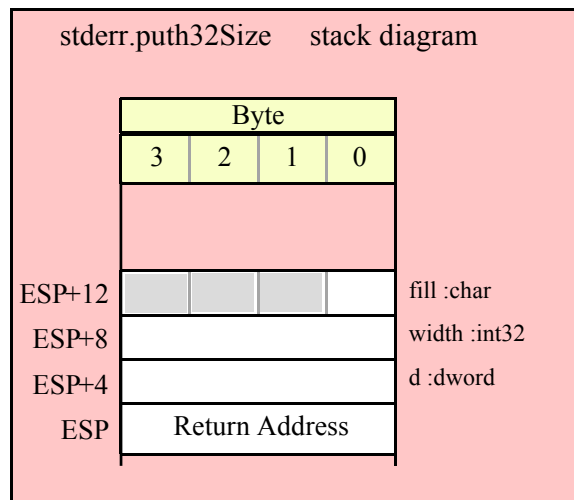
```

push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call stderr.putu32Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( eax );
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call stderr.puti32Size;

```



**stderr.puti64( q:qword );**

This function converts the 64-bit signed integer you pass as a parameter to a string and writes this string to the standard error using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

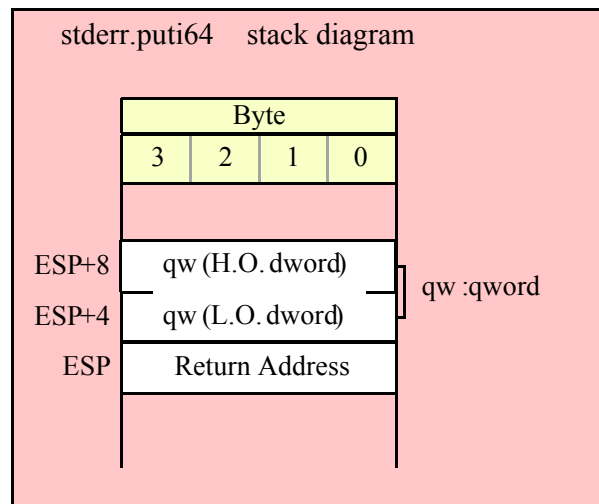
```
stderr.puti64( qwordVar );
```

HLA low-level calling sequence examples:

```

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
call stderr.puti64;

```



```
stderr.puti64Size( q:qword; width:int32; fill:char );
```

This function writes the 64-bit value you pass as a signed integer to the standard error file using the width and fill values as specified above.

HLA high-level calling sequence examples:

```
stderr.puti64Size( qwordVar, width, ' ' );
```

HLA low-level calling sequence examples:

```
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
pushd( ' ' );
call stderr.puti64Size;
```

```
push( edx ); // Assume 64-bit value in edx:eax
push( eax );
push( width );
push( ecx ); // fill char is in CL
call stderr.puti64Size;
```

```
// Assume fill char is in CH
```

```
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call stderr.puti64Size;
```

```
// Alternate method of the above
```

```
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
```

```

push( width );
sub( 4, esp );
mov( ch, [esp] );
call stderr.puti64Size;

// If the fill char is a variable and
// a register is available, try this code:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));     // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call stderr.puti64Size;

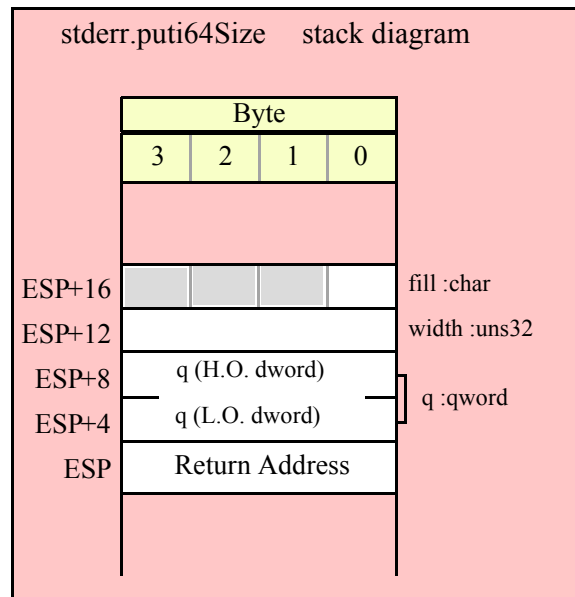
// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));     // L.O. dword last
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call stderr.puti64Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));     // L.O. dword last
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call stderr.puti64Size;

```





```
stderr.puti128( 1:1word );
```

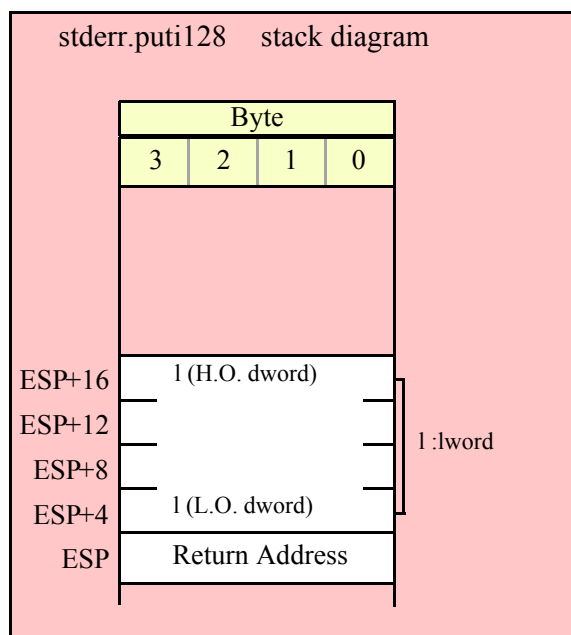
This function converts the 128-bit signed integer you pass as a parameter to a string and writes this string to the standard err using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
stderr.puti128( lwordVar );
```

HLA low-level calling sequence examples:

```
push( (type dword lwordVar[12]) ); // H.O. dword first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
call stderr.puti128;
```



```
stderr.puti128Size( 1:1word; width:int32; fill:char );
```

This function writes the 128-bit value you pass as a signed integer to the standard error device using the width and fill values as specified above.

HLA high-level calling sequence examples:

```
stderr.puti128Size( lwordVar, width, ' ' );
```

HLA low-level calling sequence examples:

```
push( (type dword lwordVar[12]) ); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
push( width );
pushd( ' ' );
```

```

call stderr.putil28Size;

// Assume fill char is in CH

push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar)); // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call stderr.putil28Size;

// Alternate method of the above

push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar)); // L.O. dword last
push( width );
sub( 4, esp );
mov( ch, [esp] );
call stderr.putil28Size;

// If the fill char is a variable and
// a register is available, try this code:

push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar)); // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call stderr.putil28Size;

// If the fill char is a variable and
// no register is available, here's one
// possibility:

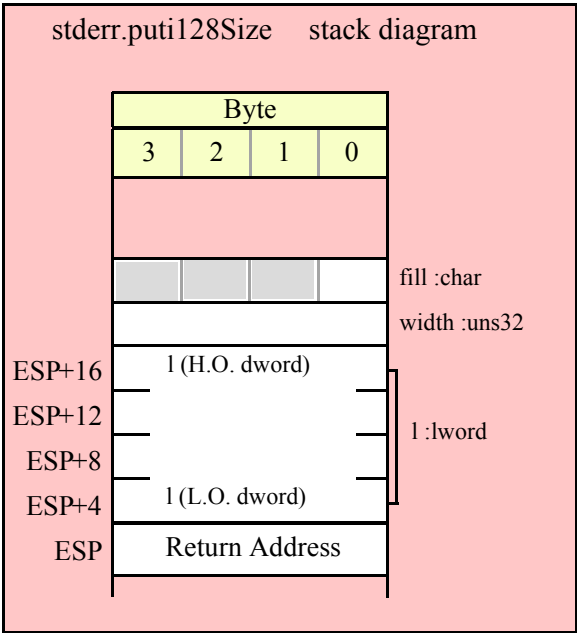
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar)); // L.O. dword last
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call stderr.putil28Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar)); // L.O. dword last
push( width );
sub( 4, esp );
push( eax );

```

```
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call stderr.puti128Size;
```



## 29.9 Unsigned Integer Output Routines

These routines convert unsigned integer values to string format and write that string to the standard error device. The `stderr.putxxxSize` functions contain width and fill parameters that let you specify the minimum field width when outputting a value.

If the absolute value of width is greater than the number of print positions the value requires, then these functions output width characters to the standard err. If width is non-negative, then these functions right-justify the value in the output field; if value is negative, then these functions left-justify the value in the output field.

These functions print the fill character as the padding value for the extra print positions.

xxxSize( value, width, fill );

Assuming “value” requires five print positions, “width” is eight, and fill is “f” then the xxxSize functions produce the string

f	f	f	V	A	L	U	E
---	---	---	---	---	---	---	---

Assuming “value” requires five print positions, “width” is minus eight, and fill is “f” then the xxxSize functions produce the string

V	A	L	U	E	f	f	f
---	---	---	---	---	---	---	---

```
stderr.putu8 ( b:byte );
```

This function converts the eight-bit unsigned integer you pass as a parameter to a string and writes this string to the standard error device using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
stderr.putu8( byteVar );
```

```
// If the character is in a register (AL):
```

```
stderr.putu8( al );
```

HLA low-level calling sequence examples:

```
// If "byteVar " is not one of the last three
// bytes on a page of memory, you can do this:
```

```
push( (type dword byteVar ) );
call stderr.putu8;
```

```
// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:
```

```
movzx( byteVar , eax ); // Assume EAX is available
push( eax );
call stderr.putu8;
```

```
// If no register is available, do something
// like the following code:
```

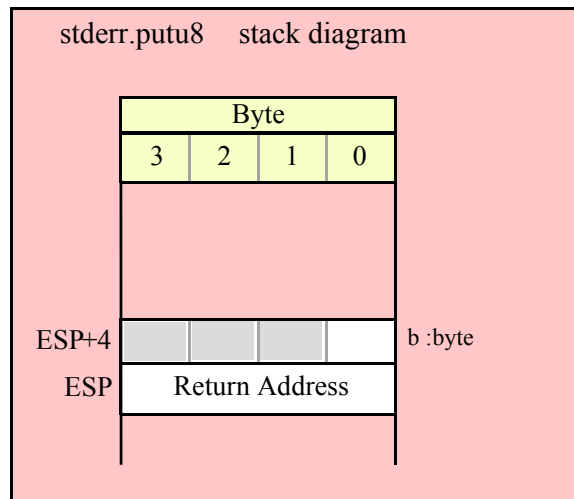
```
push( eax );
movzx( byteVar , eax );
push( eax );
call stderr.putu8;
pop( eax );
```

```
// If the character value is in al, bl, cl, or dl
// then you can use code like the following:
```

```
push( eax ); // Assume byteVar is in AL
call stderr.putu8;
```

```
// If the byte value is in ah, bh, ch, or dh
// you'll have to use code like the following:
```

```
xchg( al, ah ); // Assume byteVar is in AH
push( eax );    // It's now in AL
xchg( al, ah ); // Restore al/ah
call stderr.putu8;
```



```
stderr.putu8Size( b:byte; width:int32; fill:char );
```

This function writes the unsigned eight-bit value you pass to the standard error using the width and fill values as specified above.

HLA high-level calling sequence examples:

```
stderr.putu8Size( byteVar, width, padChar );
```

HLA low-level calling sequence examples:

```
// If "byteVar" and "padChar" are not one of the last three
// bytes on a page of memory, you can do this:
```

```
push( (type dword byteVar) );
push( width );
push( (type dword padChar) );
call stderr.putu8Size;
```

```
// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:
```

```
movzx( byteVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call stderr.putu8Size;
```

```
// If no registers are available, do something
// like the following code:
```

```
push( eax );
movzx( byteVar, eax );
push( eax );
push( width );
movzx( padChar, eax );
```

```

push( eax );
call stderr.putu8Size;
pop( eax );

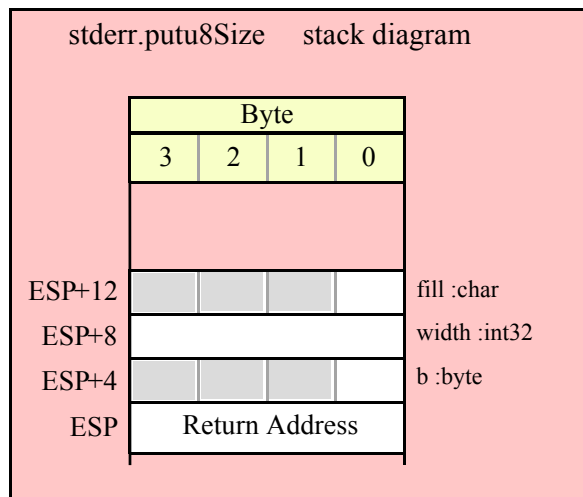
// If "byteVar" or "padChar" are in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:

push( eax );    // Assume byteVar is in AL
push( width );
push( ebx );    // Assume padChar is in BL
call stderr.putu8Size;

// Do the following if the characters are
// in AH, BH, CH, or DH:

xchg( al, ah );    // Assume byteVar is in AH
xchg( bl, bh );    // Assume padChar is in BH
push( eax );
push( width );
push( ebx );
xchg( al, ah );
xchg( bl, bh );
call stderr.putu8Size;

```



**stderr.putu16( w:word );**

This function converts the 16-bit unsigned integer you pass as a parameter to a string and writes this string to the standard error device using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
stderr.putu16( wordVar );
```

```
// If the word is in a register (AX):
```

```
stderr.putu16( ax );
```

HLA low-level calling sequence examples:

```

// If "wordVar " is not one of the last three
// bytes on a page of memory, you can do this:

push( (type dword wordVar) );
call stderr.putul6;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( wordVar, eax ); // Assume EAX is available
push( eax );
call stderr.putul6;

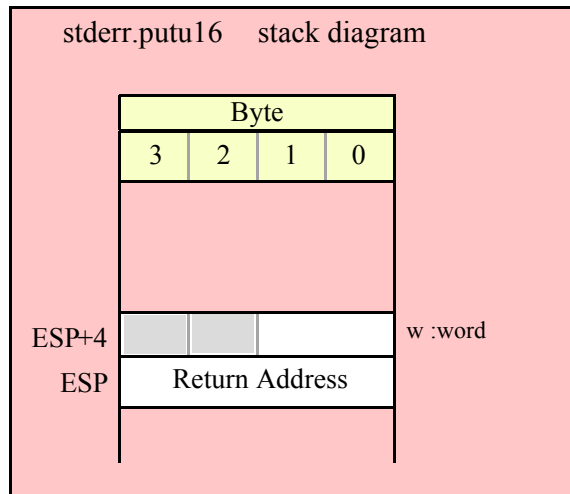
// If no register is available, do something
// like the following code:

push( eax );
movzx( wordVar, eax );
push( eax );
call stderr.putul6;
pop( eax );

// If the word value is in a 16-bit register
// then you can use code like the following:

push( eax ); // Assume wordVar is in AX
call stderr.putul6;

```



```
stderr.putul6Size( w:word; width:int32; fill:char );
```

This function writes the unsigned 16-bit value you pass to the standard err using the width and fill values as specified above.

HLA high-level calling sequence examples:

```
stderr.putul6Size( wordVar, width, padChar );
```

HLA low-level calling sequence examples:

```
// If "wordVar" and "padChar" are not one of the last three
// bytes on a page of memory, you can do this:

push( (type dword wordVar) );
push( width );
push( (type dword padChar) );
call stderr.putul6Size;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( wordVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call stderr.putul6Size;

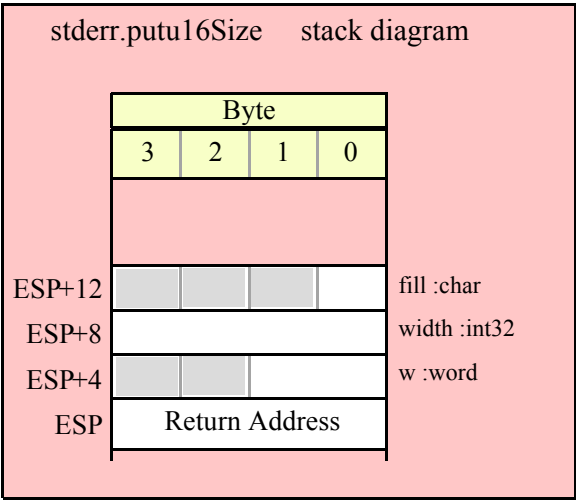
// If no registers are available, do something
// like the following code:

push( eax );
movzx( wordVar, eax );
push( eax );
push( width );
movzx( padChar, eax );
push( eax );
call stderr.putul6Size;
pop( eax );

// If "wordVar" is in a 16-bit register
// and "padChar" is in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:

push( eax ); // Assume wordVar is in AX
push( width );
push( ebx ); // Assume padChar is in BL
call stderr.putul6Size;
```





**stderr.putu32( d:dword );**

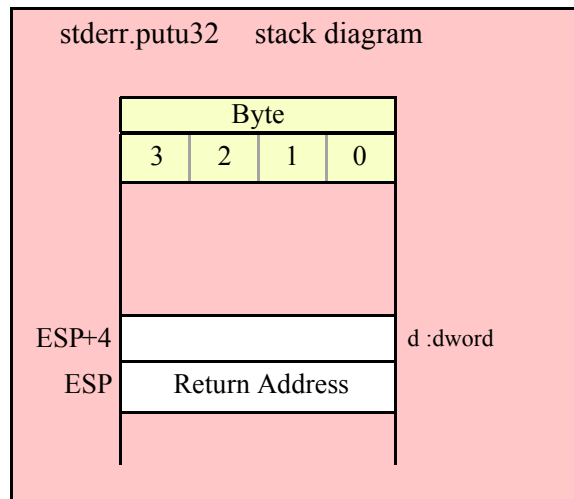
This function converts the 32-bit unsigned integer you pass as a parameter to a string and writes this string to the standard error device using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
stderr.putu32( dwordVar );  
  
// If the dword is in a register (EAX):  
  
stderr.putu32( eax );
```

HLA low-level calling sequence examples:

```
push( dwordVar );  
call stderr.putu32;  
  
push( eax );  
call stderr.putu32;
```



```
stderr.putu32Size( d:dword; width:int32; fill:char );
```

This function writes the unsigned 32-bit value you pass to the standard err using the width and fill values as specified above.

HLA high-level calling sequence examples:

```
stderr.putu32Size( dwordVar, width, ' ' );

// If the dword is in a register (EAX):

stderr.putu32Size( eax, width, cl );
```

HLA low-level calling sequence examples:

```
push( dwordVar );
push( width );
pushd( ' ' );
call stderr.putu32Size;

push( eax );
push( width );
push( ecx ); // fill char is in CL
call stderr.putu32Size;

// Assume fill char is in CH

push( eax );
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call stderr.putu32Size;

// Alternate method of the above

push( eax );
push( width );
```

```
sub( 4, esp );
mov( ch, [esp] );
call stderr.putu32Size;

// If the fill char is a variable and
// a register is available, try this code:

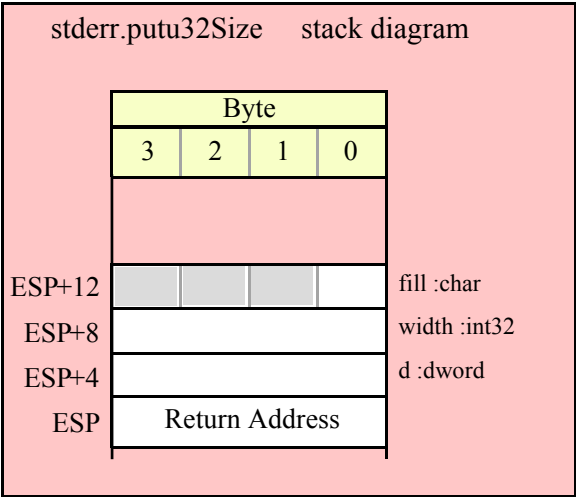
push( eax );
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call stderr.putu32Size;

// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( eax );
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call stderr.putu32Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( eax );
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call stderr.putu32Size;
```



```
stderr.putu64( q:qword );
```

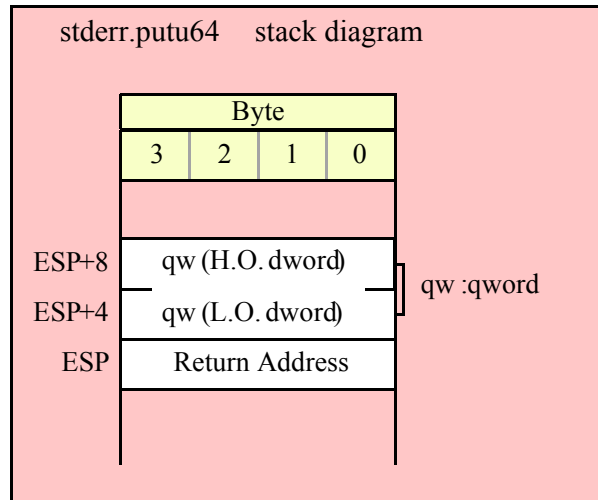
This function converts the 64-bit unsigned integer you pass as a parameter to a string and writes this string to the standard error device using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
stderr.putu64( qwordVar );
```

HLA low-level calling sequence examples:

```
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
call stderr.putu64;
```



```
stderr.putu64Size( q:qword; width:int32; fill:char );
```

This function writes the unsigned 64-bit value you pass to the error output using the width and fill values as specified above.

HLA high-level calling sequence examples:

```
stderr.putu64Size( qwordVar, width, ' ' );
```

HLA low-level calling sequence examples:

```
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
pushd( ' ' );
call stderr.putu64Size;
```

```
push( edx ); // Assume 64-bit value in edx:eax
push( eax );
push( width );
push( ecx ); // fill char is in CL
call stderr.putu64Size;
```

```
// Assume fill char is in CH
```

```
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
```

```

xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call stderr.putu64Size;

// Alternate method of the above

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));      // L.O. dword last
push( width );
sub( 4, esp );
mov( ch, [esp] );
call stderr.putu64Size;

// If the fill char is a variable and
// a register is available, try this code:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));      // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call stderr.putu64Size;

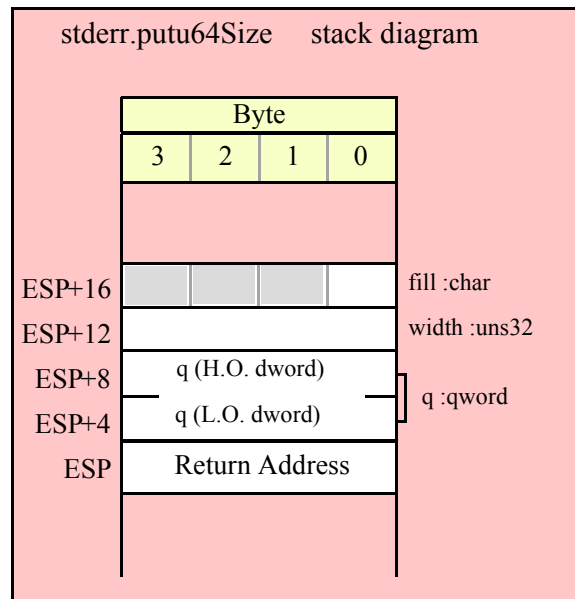
// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));      // L.O. dword last
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call stderr.putu64Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));      // L.O. dword last
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call stderr.putu64Size;

```



```
stderr.putu128( l:1word );
```

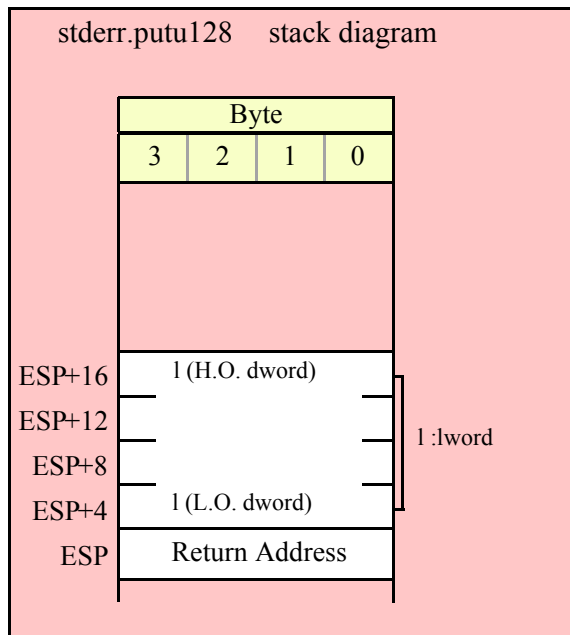
This function converts the 128-bit unsigned integer you pass as a parameter to a string and writes this string to the standard err using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
stderr.putu128( lwordVar );
```

HLA low-level calling sequence examples:

```
push( (type dword lwordVar[12]) ); // H.O. dword first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar)); // L.O. dword last
call stderr.putul28;
```



**stderr.putu128Size( 1:lword; width:int32; fill:char )**

This function writes the unsigned 128-bit value you pass to the standard err using the width and fill values as specified above.

HLA high-level calling sequence examples:

```
stderr.putu128Size( lwordVar, width, ' ' );
```

HLA low-level calling sequence examples:

```
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));    // L.O. dword last
push( width );
pushd( ' ' );
call stderr.putu128Size;
```

```
// Assume fill char is in CH
```

```
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));    // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call stderr.putu128Size;
```

```
// Alternate method of the above
```

```
push( (type dword lwordVar[12])); // Push H.O. word first
```

```

push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
push( width );
sub( 4, esp );
mov( ch, [esp] );
call stderr.putul28Size;

// If the fill char is a variable and
// a register is available, try this code:

push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call stderr.putul28Size;

// If the fill char is a variable and
// no register is available, here's one
// possibility:

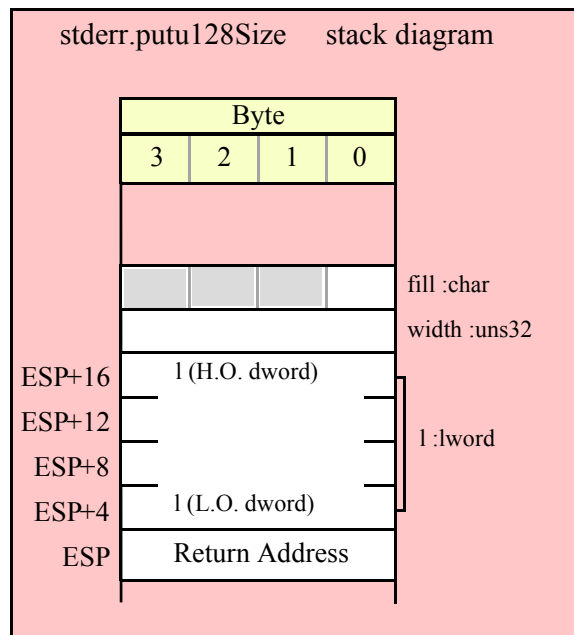
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
push( width );
push( (type dword fillChar) );      // Chance of page crossing!
call stderr.putul28Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call stderr.putul28Size;

```





## 29.10 Floating Point Output Routines

The HLA standard error module provides several procedures you can use to write floating point values to the standard error device. The following subsections describe these routines.

### 29.10.1 Real Output Using Scientific Notation

The floating point numeric output routines translate the three different binary floating point formats to their string representation and then write this string to the standard error. There are two generic classes of these routines: those that convert their values to exponential/scientific notation and those that convert their string to a decimal form.

The `stderr.pute80`, `stderr.pute64`, and `stderr.pute32` routines convert their values to a string using scientific notation. These three routines each have two parameters: the value to output and the field width of the result. These routines produce a string with the following format:

s	i	.	f	f	f	f	f	E	±	x
---	---	---	---	---	---	---	---	---	---	---

s is a space for positive values, '-' for negative values  
i represents the integer portion of the mantissa  
ffff represents the fractional portion of the mantissa  
x is one or more base-10 exponent digits.

```
stderr.pute32( r:real32; width:uns32 );
```

This function writes the 32-bit single precision floating point value passed in `r` to the standard err using scientific/exponential notation. This procedure prints the value using width print positions in the output. `width` should have a minimum value of five for real numbers in the range  $1e-9..1e+9$  and a minimum value of six for all other values. Note that 32-bit extended precision floating point values support about 6-7 significant digits. So a width value that yeilds more than seven mantissa digits will produce garbage output in the low order digits of the number.

HLA high-level calling sequence examples:

```
stderr.pute32( r32Var, width );

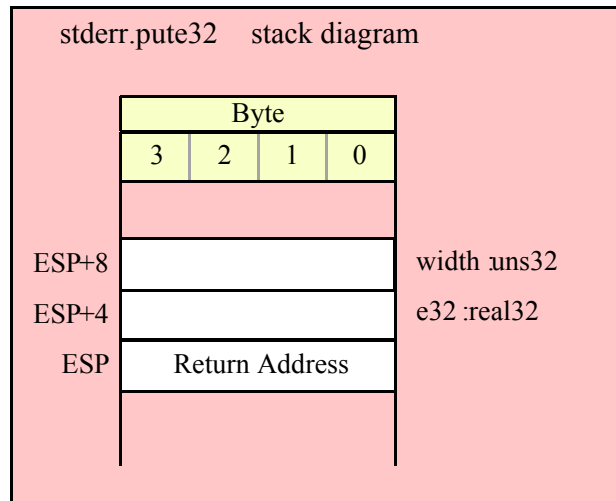
// If the real32 value is in an FPU register (ST0):

var
    r32Temp:real32;
    .
    .
    .
fstp( r32Temp );
stderr.pute32( r32Temp, 12 );
```

HLA low-level calling sequence examples:

```
push( (type dword r32Var) );
push( width );
call stderr.pute32;

sub( 4, esp );
fstp( (type real32 [esp]) );
pushd( 12 );
call stderr.pute32;
```



**stderr.pute64( r:real64; width:uns32 );**

This function writes the 64-bit double precision floating point value passed in *r* to the standard error using scientific/exponential notation. This procedure prints the value using *width* print positions in the output. *width* should have a minimum value of five for real numbers in the range 1e-9..1e+9 and a minimum value of six for all other values. Note that 64-bit double precision floating point values support about 15 significant digits. So a *width* value that yields more than 15 mantissa digits will produce garbage output in the low order digits of the number.

HLA high-level calling sequence examples:

```
stderr.pute64( r64Var, width );

// If the real64 value is in an FPU register (ST0):
```

```

var
    r64Temp:real64;
    .
    .
    .
fstp( r64Temp );
stderr.pute64( r64Temp, 12 );

```

HLA low-level calling sequence examples:

```

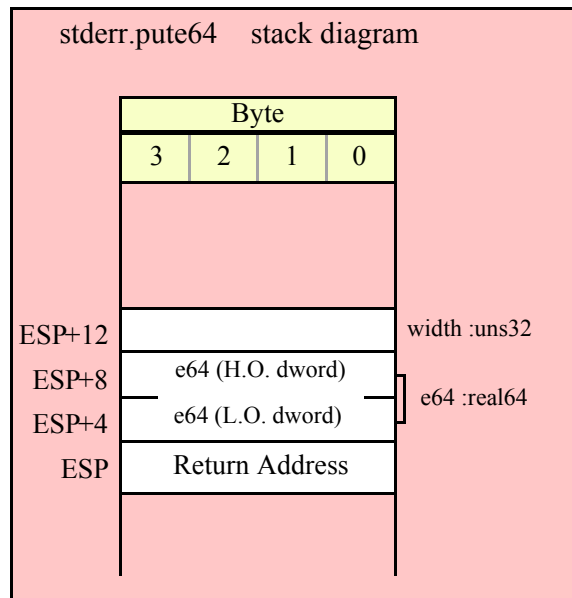
push( (type dword r64Var[4]));
push( (type dword r64Var[0]));
push( width );
call stderr.pute64;

```

```

sub( 8, esp );
fstp( (type real64 [esp]) );
pushd( 12 );
call stderr.pute64;

```



```
stderr.pute80( r:real80; width:uns32 );
```

This function writes the 80-bit extended precision floating point value passed in *r* to the standard error device using scientific/exponential notation. This procedure prints the value using *width* print positions in the standard err. *width* should have a minimum value of five for real numbers in the range 1e-9..1e+9 and a minimum value of six for all other values. Note that 80-bit extended precision floating point values support about 18 significant digits. So a *width* value that yeilds more than 18 mantissa digits will produce garbage output in the low order digits of the number.

HLA high-level calling sequence examples:

```
stderr.pute80( r80Var, width );
```

```
// If the real80 value is in an FPU register (ST0):
```

```
var
```

```

    r80Temp:real80;
    .
    .
    .
    fstp( r80Temp );
    stderr.pute80( r80Temp, 12 );

```

HLA low-level calling sequence examples:

```

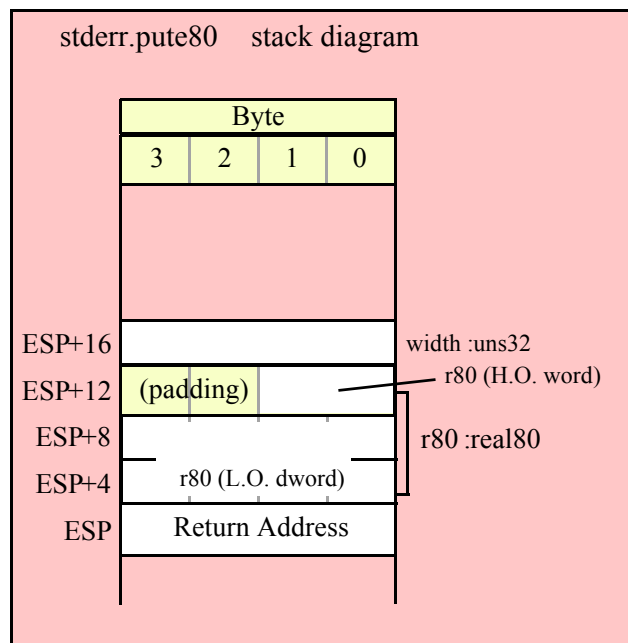
pushw( 0 ); // A word of padding.
push( (type word r80Var[8]));
push( (type dword r80Var[4]));
push( (type dword r80Var[0]));
push( width );
call stderr.pute80;

```

```

sub( 12, esp );
fstp( (type real80 [esp]) );
pushd( 12 );
call stderr.pute80;

```



## 29.10.2 Real Output Using Decimal Notation

Although scientific (exponential) notation is the most general display format for real numbers, real numbers you display in this format are very difficult to read. Therefore, the HLA `stderr` module also provides a set of functions that output real values using the decimal representation. Although you cannot (practically) use these decimal output routines for all real values, they are applicable to a wide variety of common numbers you will use in your programs.

These functions require four parameters: the real value to convert, the width of the converted value, the number of digit positions to the right of the decimal point, and a padding character. These functions write their values using the following string format:

s	i	i	i	.	f	f	f	f	f	f
---	---	---	---	---	---	---	---	---	---	---

s is a space for positive values, '-' for negative values  
i represents the integer portion of the mantissa  
ffff represents the fractional portion of the mantissa

This procedure writes a 32-bit single precision floating point value to the standard error as a string. The string consumes exactly width characters in the output. If the numeric output, using the specified number of positions to the right of the decimal point, is sufficiently small that the string representation would be less than width characters, then this procedure uses the value of pad as the padding character to fill the output with width characters. The number is right-justified within the output field (that is, this function prints the padding characters before the string representation of the number).

HLA high-level calling sequence examples:

```
stderr.putr32( r32Var, width, decpts, fill );
stderr.putr32( r32Var, 10, 2, '*' );

// If the real32 value is in an FPU register (ST0):

var
    r32Temp:real32;
.
.
.
fstp( r32Temp );
stderr.putr32( r32Temp, 12, 2, al );
```

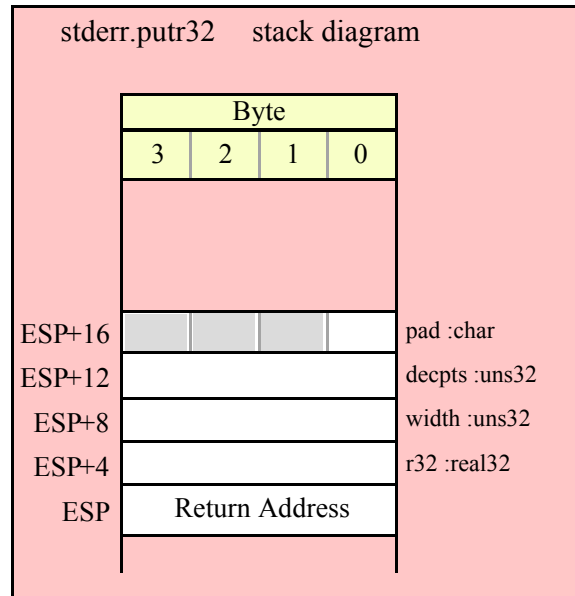
HLA low-level calling sequence examples:

```
push( (type dword r32Var) );
push( width );
push( decpts );
movzx( fill, eax );
push( eax );
call stderr.putr32;

push( (type dword r32Var) );
push( width );
push( decpts );
pushd( (type dword fill) ); // If no memory fault possible
call stderr.putr32;

sub( 4, esp );
fstp( (type real32 [esp]) );
pushd( 12 );
sub( 4, esp );
push( eax );
movzx( fill, eax ); // If memory fault were possible
mov( eax, [esp+4] ); // in above code.
pop( eax );
```

```
call stderr.putr32;
```



```
stderr.putr64( r:real64; width:uns32; decpts:uns32; pad:char);
```

This procedure writes a 64-bit double precision floating point value to the standard error device as a string. The string consumes exactly width characters in the output. If the numeric output, using the specified number of positions to the right of the decimal point, is sufficiently small that the string representation would be less than width characters, then this procedure uses the value of pad as the padding character to fill the output with width characters.

HLA high-level calling sequence examples:

```
stderr.putr64( r64Var, width, decpts, fill );
stderr.putr64( r64Var, 10, 2, '*' );

// If the real64 value is in an FPU register (ST0):

var
    r64Temp:real64;
.
.
.
fstp( r64Temp );
stderr.putr64( r64Temp, 12, 2, al );
```

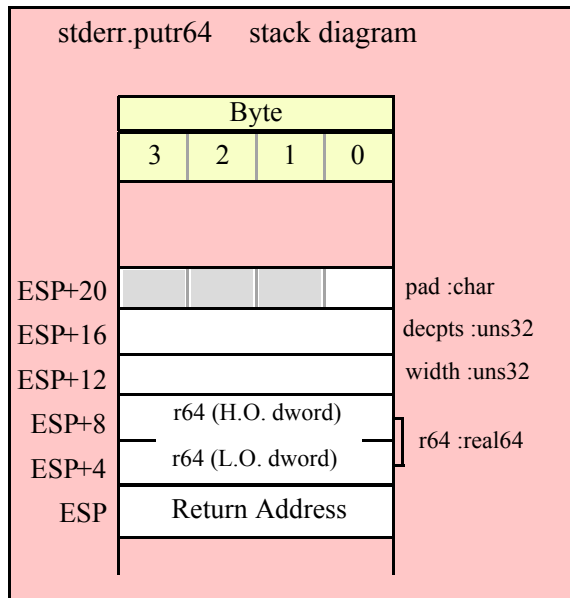
HLA low-level calling sequence examples:

```
push( (type dword r64Var[4]) );
push( (type dword r64Var) );
push( width );
push( decpts );
movzx( fill, eax );
push( eax );
call stderr.putr64;

push( (type dword r64Var[4]) );
push( (type dword r64Var) );
```

```
push( width );
push( decpts );
pushd( (type dword fill) ); // If no memory fault possible
call stderr.putr64;

sub( 8, esp );
fstp( (type real64 [esp]) );
pushd( 12 );
sub( 4, esp );
push( eax );
movzx( fill, eax ); // If memory fault were possible
mov( eax, [esp+4] ); // in above code.
pop( eax );
call stderr.putr64;
```



```
stderr.putr80( r:real80; width:uns32; decpts:uns32; pad:char);
```

This procedure writes an 80-bit extended precision floating point value to the output as a string. The string consumes exactly width characters in the output. If the numeric output, using the specified number of positions to the right of the decimal point, is sufficiently small that the string representation would be less than width characters, then this procedure uses the value of pad as the padding character to fill the output with width characters.

HLA high-level calling sequence examples:

```
stderr.putr80( r80Var, width, decpts, fill );
stderr.putr80( r80Var, 10, 2, '*' );

// If the real80 value is in an FPU register (ST0):

var
    r80Temp:real80;
.
.
.
fstp( r80Temp );
stderr.putr80( r80Temp, 12, 2, al );
```

HLA low-level calling sequence examples:

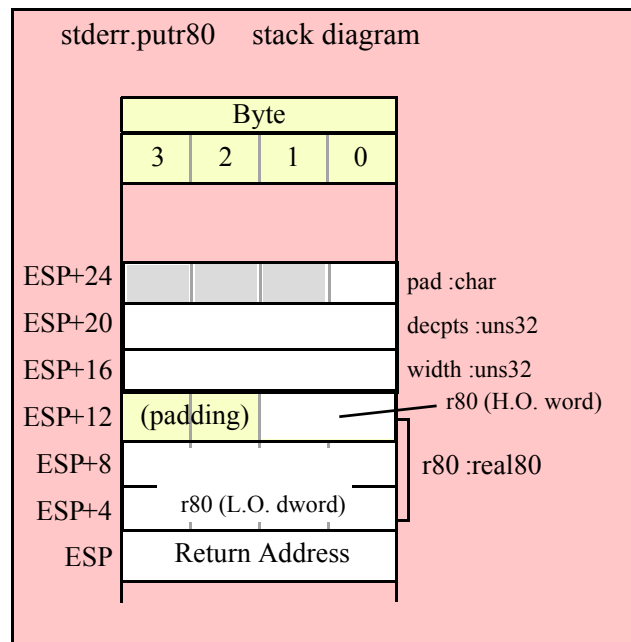
```

pushw( 0 );
push( (type word r80Var[8]) );
push( (type dword r80Var[4]) );
push( (type dword r80Var) );
push( width );
push( decpts );
movzx( fill, eax );
push( eax );
call stderr.putr80;

pushw( 0 );
push( (type word r80Var[8]) );
push( (type dword r80Var[4]) );
push( (type dword r80Var) );
push( width );
push( decpts );
pushd( (type dword fill) ); // If no memory fault possible
call stderr.putr80;

sub( 12, esp );
fstp( (type real80 [esp]) );
pushd( 12 );
sub( 4, esp );
push( eax );
movzx( fill, eax ); // If memory fault were possible
mov( eax, [esp+4] ); // in above code.
pop( eax );
call stderr.putr80;

```





## 29.11 Generic Error Output Routine

```
stderr.put( list_of_items );
```

`stderr.put` is a macro that automatically invokes an appropriate `stderr` output routine based on the type of the parameter(s) you pass it. This is a very convenient output routine and is probably the `stderr` output call you will use most often in your programs. Keep in mind that this macro is not a single function call; instead, HLA translates this macro into a sequence of calls to procedures like `stderr.putu32`, `stderr.puts`, etc.

`stderr.put` is a macro that provides a flexible syntax for outputting data to the standard error device. This macro allows a variable number of parameters. For each parameter present in the list, `stderr.put` will call the appropriate routine to emit that data, according to the type of the parameter. Parameters may be constants, registers, or memory locations. You must separate each macro parameter with a comma.

Here is an example of a typical invocation of `stderr.put`:

```
stderr.put( "I=", i, " j=", j, nl );
```

The above is roughly equivalent to

```
stderr.puts( "I=" );  
stderr.putu32( i );  
stderr.puts( " j=" );  
stderr.putu32( j );  
stderr.newln();
```

This assumes, of course, that `i` and `j` are `int32` variables.

The `stderr.put` macro also lets you specify the minimum field width for each parameter you specify. To print a value using a minimum field width, follow the object you wish to print with a colon and the value of the minimum field width. The previous example, using field widths, could look like the following:

```
stderr.put( "I=", i:2, " j=", j:5, nl );
```

Although this example used the literal decimal constants two and five for the field widths, keep in mind that register values and memory value (integers, anyway) are perfectly legal here.

For floating point numbers you wish to display in decimal form, you can specify both the minimum field width and the number of digits to print to the right of the decimal point by using the following syntax:

```
stderr.put( "Real value is ", f:10:3, nl );
```

The `stderr.put` macro can handle all the basic primitive types, including boolean, unsigned (8, 16, 32, 64, 128), signed (8, 16, 32, 64, 128), character, character set, real (32, 64, 80), string, and hexadecimal (byte, word, dword, qword, lword).

There is a known "design flaw" in the `stderr.put` macro. You cannot use it to print HLA intermediate variables (i.e., non-local VAR objects). The problem is that HLA's syntax for non-local accesses takes the form "`reg32:varname`" and `stderr.put` cannot determine if you want to print `reg32` using `varname` print positions versus simply printing the non-local `varname` object. If you want to display non-local variables you must copy the non-local object into a register, a static variable, or a local variable prior to using `stderr.put` to print it. Of course, there is no problem using the other `stderr.putXXXX` functions to display non-local VAR objects, so you can use those as well.



## 30 The Standard Input Module (stdin.hhf)

This unit contains routines that read data from the standard input device.

**Note:** be sure to read the chapter on "Passing Parameters to Standard Library Routines" (parmpassing.rtf) before reading this chapter. Whenever you request input, by calling one of the following input routines, the Standard Library routines first check to see if there is any data available in an internal buffer. If so, the routines read the data from the buffer; if not, the routines fill the buffer by reading a line of text from the Standard Input Device. Once a line is read, the routine will read its data from the newly acquired buffer. Additional calls to the standard input routines continue to read their data from this same buffer until the input line is exhausted, at which point the library routines will read more data from the Standard Input Device.

**A Note About Thread Safety:** Because the standard input device is a single resource, you will get inconsistent results if multiple threads attempt to read from the standard input device simultaneously. The HLA standard library stdin module does not attempt to synchronize thread access to the standard input device. If you are going to be reading from the standard input from multiple threads, it is your responsibility to ensure that the threads use properly synchronized access to this resource.

**Note about stack diagrams:** this documentation includes stack diagrams for those functions that pass parameters on the stack. To conserve space, this documentation does not include a stack diagram for any function that does not pass data on the stack (that is, only a return address appears on the stack).

**A Note About the FPU:** The Standard Library code makes occasional use of the FPU, particularly when converting between real and string formats and when computing certain mathematical functions. You should exercise caution when using MMX instructions in a program that makes use of the Standard Library. In particular, you should ensure that you are always in FPU mode (by executing an EMMS instruction) after you are finished using MMX instructions. Better yet, you should avoid the MMX instruction set altogether and use the improved SSE instruction set that accomplishes the same tasks (and doesn't disturb the FPU).

### 30.1 Conversion Format Control

When reading numeric data from the standard input, the stdin functions use an internal delimiters character set to determine which characters may legally end a sequence of numeric digits. You can change the complexion of this character set using the `conv.getDelimiters` and `conv.setDelimiters` functions. Please refer to their documentation in the `conv.rtf` file for more details.

### 30.2 File I/O Routines and the Standard Output Handle

The standard input routines are basically a thin layer over the top of the fileio routines (see the fileio documentation for a complete description of those routines). Indeed, if you obtain the standard input handle, you can read data from the standard input device by passing this handle to a fileio function. Because the fileio module provides a slightly richer set of routines, there are a few instances where you might want to do this. You might also want to write a generic input function that expects a file handle and then pass it the standard input device file handle so that the function reads its input from the console (or other standard input device) rather than to some file. In any case, just be aware that it is perfectly reasonable to call fileio functions to read data from the standard input device.

```
stdin.handle; @returns( "eax" );
```

This routine returns the handle of the Standard Input Device in the EAX register.

### 30.3 Standard Input Routines

The HLA Standard Library provides a complementary set of standard input routines. These routines behave in a fashion quite similar to the `stdin.XXXX` routines. See those routines for additional examples of these procedures.

## 30.4 General Standard Input Routines

```
stdin.read( var buffer:byte; count:uns32 )
```

This routine reads a sequence of count bytes from the standard input device, storing the bytes into memory at the address specified by buffer.

HLA high-level calling sequence examples:

```
stdin.read( buffer, count );
stdin.read( [eax], 1024 );
```

HLA low-level calling sequence examples:

```
// If buffer is a static variable:
```

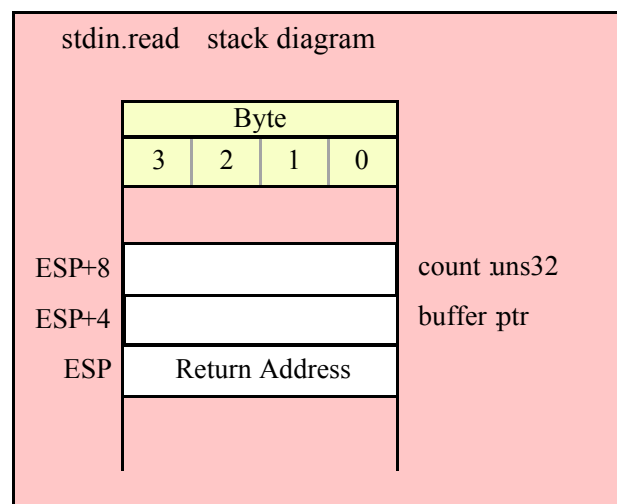
```
pushd( &buffer );
push( count );
call stdin.read;
```

```
// If buffer is not static, 32-bit register available:
```

```
lea( eax, buffer );
push( eax );
push( count );
call stdin.read;
```

```
// If buffer is not static, no register available:
```

```
sub( 4, esp );
push( eax );
lea( eax, buffer );
mov( eax, [esp+4] );
pop( eax );
push( count );
call stdin.read;
```



**stdin.readLine;**

This routine flushes the current input buffer and immediately reads a new line of text from the user.

HLA high-level calling sequence examples:

```
stdin.readLine();
```

HLA low-level calling sequence examples:

```
call stdin.readLine;
```

```
stdin.eoln; @returns( "al" );  
stdin.eoln2; @returns( "al" );
```

These functions return true if the input buffer is at the end of the current line. The `stdin.eoln2` function will first remove any delimiter characters from the input buffer before testing for the end of the current line. These functions return true (1) or false (0) in the AL/EAX register.

These functions do not force a new line of input on the next `stdin.getXX` operation. I.e., if you read a string after `stdin.eoln` returns true, you will get the empty string as the result. Call `stdin.readLine` to force the input of a new line.

HLA high-level calling sequence examples:

```
stdin.eoln();  
mov( al, eolnVar );
```

HLA low-level calling sequence examples:

```
call stdin.eoln;  
mov( al, eolnVar );
```

**stdin.flushInput;**

This routine flushes the internal buffer. The next call to a Standard Library input routine will force the system to read a new line of text from the user. All current data in the internal input buffer is lost.

Please note that this routine does not immediately force the input of a new line of text from the user unless the internal buffer is already empty. If the internal buffer is empty and you call this routine, it will read a new line of text from the user and then flush this text from the internal buffer.

HLA high-level calling sequence examples:

```
stdin.flushInput();
```

HLA low-level calling sequence examples:

```
call stdin.flushInput;
```

## 30.5 Character and String Input Routines

The following functions read character data from an input file specified by filevar. Note that HLA's stdin module does not provide the ability to read character set data directly from the user. However, you can always read a string and then convert that string to a character set using the appropriate function in the cset module.

```
stdin.peekc; @returns( "al" );
```

This routine returns the character character from the standard input device without actually "reading" that character. That is, after a call to stdin.peekc, the next call to stdin.getc will return the same character as the one stdin.peekc returns. A call to stdin.peekc does not force the input of a new line of text. If the current input buffer is empty, calls to stdin.peekc return zero in the AL register. This routine returns the character in the AL register and it returns zeros in the upper three bytes of EAX.

```
stdin.getc(); @returns( "al" );
```

This function reads a single character from the standard input device and returns that character in the AL register.

```
stdin.gets( s:string );
```

This function reads a sequence of characters from the standard input through to the next end of line sequence and stores these characters (without the end of line sequence) into the string variable you pass as a parameter. Before calling this routine, you must allocate sufficient storage for the string. If stdin.gets attempts to read a larger string than the string's MaxStrLen value, stdin.gets raises a string overflow exception.

Note that this function does not store the end of line sequence into the string, though it does consume the end of line sequence. The next character a stdin function will read from the standard input will be the first character of the following line.

If the standard input is at the end of some line of text, then stdin.gets consumes the end of line and stores the empty string into the s parameter.

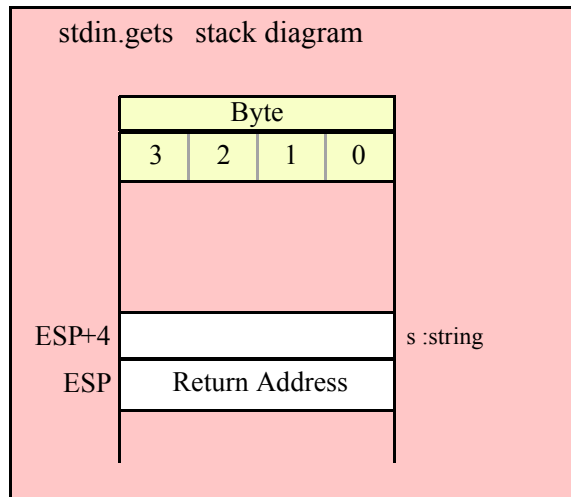
HLA high-level calling sequence examples:

```
stdin.gets( inputStr );
stdin.gets( eax ); // EAX contains string value
```

HLA low-level calling sequence examples:

```
push( inputStr );
call stdin.gets;

push( eax );
call stdin.gets;
```



```
stdin.a_gets(); @returns( "eax" );
```

Like `stdin.gets`, this function also reads a string from the standard input. However, rather than storing the string data into a string you supply, this function allocates storage for the string on the heap and returns a pointer to this string in the EAX register. You code should call `strfree` to release this storage when you're done with the string data.

The `stdin.a_gets` function imposes a line length limit of 4,096 characters. If this is a problem, you should modify the source code for this function to raise the limit. This function raises an exception if you attempt to read a line longer than this internal limit.

HLA high-level calling sequence examples:

```
stdin.a_gets();
mov( eax, inputStr );
```

HLA low-level calling sequence examples:

```
call stdin.a_gets;
mov( eax, inputStr );
```

## 30.6 Hexadecimal Input Routines

The hexadecimal input routines read a numeric value from the standard input in hexadecimal format. Except for `stdin.geth128`, they return their results in one (or two) registers (`stdin.geth128` returns its value in a pass-by-reference parameter).

```
stdin.geth8(); @returns( "al" );
```

This function reads an eight-bit hexadecimal integer in the range 0..\$FF from the standard input device. The number may begin with any number of delimiter characters (see the `conv.setDelimiter` and `conv.getDelimiter` functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "\$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character. This function allows underscores in the interior of the number. The `stdin.geth` function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..\$FF. This function returns the binary form of the value in the AL register (zero extended into EAX, so you may use EAX if it is more convenient to do so).

HLA high-level calling sequence examples:

```
stdin.geth8();
mov( al, h8Var );
```

HLA low-level calling sequence examples:

```
call stdin.geth8;
mov( al, h8Var );
```

**stdin.geth16(); @returns( "ax" );**

This function reads a 16-bit hexadecimal integer in the range 0..\$FFFF from the standard input. The number may begin with any number of delimiter characters (see the `conv.setDelimiter` and `conv.getDelimiter` functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "\$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character. This function allows underscores in the interior of the number. The `stdin.geth16` function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..\$FFFF. This function returns the binary form of the value in the AX register (zero-extended into EAX).

HLA high-level calling sequence examples:

```
stdin.geth16();
mov( ax, h16Var );
```

HLA low-level calling sequence examples:

```
call stdin.geth16;
mov( ax, h16Var );
```

**stdin.geth32(); @returns( "eax" );**

This function reads a 32-bit hexadecimal integer in the range 0..\$FFFF\_FFFF from the standard input device. The number may begin with any number of delimiter characters (see the `conv.setDelimiter` and `conv.getDelimiter` functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "\$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character. This function allows underscores in the interior of the number. The `stdin.geth32` function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..\$FFFF\_FFFF. This function returns the binary form of the value in the EAX register.

HLA high-level calling sequence examples:

```
stdin.geth32();
mov( eax, h32Var );
```

HLA low-level calling sequence examples:

```
call stdin.geth32;
mov( eax, h32Var );
```



**stdin.geth64();**

This function reads a 64-bit hexadecimal integer in the range 0..\$FFFF\_FFFF\_FFFF\_FFFF from the standard input device. The number may begin with any number of delimiter characters (see the conv.setDelimiter and conv.getDelimiter functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "\$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character. This function allows underscores in the interior of the number. The stdin.geth64 function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..\$FFFF\_FFFF\_FFFF\_FFFF. This function returns the 64-bit result in the EDX:EAX register pair.

HLA high-level calling sequence examples:

```
stdin.geth64();
mov( edx, (type dword h64Var[4]) );
mov( eax, (type dword h64Var[0]) );
```

HLA low-level calling sequence examples:

```
call stdin.geth64;
mov( edx, (type dword h64Var[4]) );
mov( eax, (type dword h64Var[0]) );
```

**stdin.geth128( var dest:lword );**

This function reads a 128-bit hexadecimal integer in the range zero through \$FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF from the standard input device. The number may begin with any number of delimiter characters (see the conv.setDelimiter and conv.getDelimiter functions for details on the delimiter characters) followed by a string of one or more hexadecimal digits. Note that the value may not have a leading "\$" unless you add this character to the delimiter character set. The number must end with a valid delimiter character. This function allows underscores in the interior of the number. The stdin.geth128 function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..\$FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF\_FFFF. This function stores the 128-bit result into the variable you pass as a reference parameter.

HLA high-level calling sequence examples:

```
stdin.geth128( lwordVar );
```

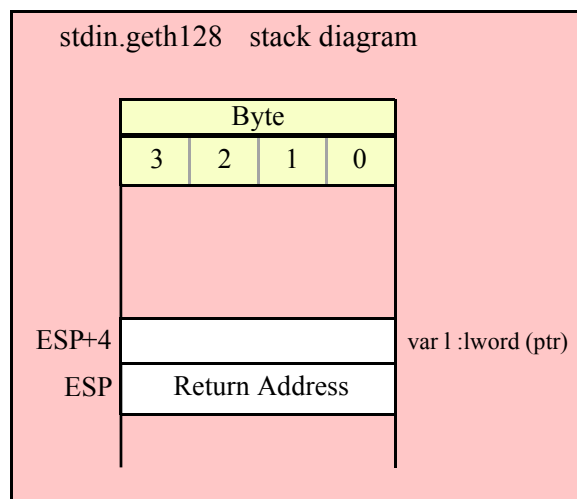
HLA low-level calling sequence examples:

```
// If lwordVar is a static variable:

pushd( &lwordVar );
call stdin.geth128;

// If lwordVar is a not static variable
// and a 32-bit register is available:

lea( eax, lwordVar ); // Assume EAX is available
push( eax );
call stdin.geth128;
```



## 30.7 Signed Integer Input Routines

```
stdin.geti8(); @returns( "al" );
```

This function reads a signed eight-bit decimal integer in the range -128..+127 from the standard input. The number may begin with any number of delimiter characters (see the `conv.setDelimiter` and `conv.getDelimiter` functions for details on the delimiter characters) followed by an optional minus sign and a string of one or more decimal digits. The number must end with a valid delimiter character. This function allows underscores in the interior of the number. The `stdin.geti8` function raises an appropriate exception if the input violates any of these rules or the value is outside the range -128..+127. This function returns the binary form of the integer in the AL register (signed extended into EAX).

HLA high-level calling sequence examples:

```
stdin.geti8();
mov( al, i8Var );
```

HLA low-level calling sequence examples:

```
call stdin.geti8;
mov( al, i8Var );
```

```
stdin.geti16(); @returns( "ax" );
```

This function reads a signed 16-bit decimal integer in the range -32768..+32767 from the standard input device. The number may begin with any number of delimiter characters (see the `conv.setDelimiter` and `conv.getDelimiter` functions for details on the delimiter characters) followed by an optional minus sign and a string of one or more decimal digits. The number must end with a valid delimiter character. This function allows underscores in the interior of the number. The `stdin.geti16` function raises an appropriate exception if the input violates any of these rules or the value is outside the range -32768..+32767. This function returns the binary form of the integer in the AX register (signed extended into EAX).

HLA high-level calling sequence examples:

```
stdin.geti16();
mov( ax, i16Var );
```

HLA low-level calling sequence examples:

```
call stdin.geti16;
mov( ax, i16Var );
```

**stdin.geti32(); @returns( "eax" );**

This function reads a signed 32-bit decimal integer in the (approximate) range  $\pm 2$  Billion from the standard input device. The number may begin with any number of delimiter characters (see the `conv.setDelimiter` and `conv.getDelimiter` functions for details on the delimiter characters) followed by an optional minus sign and a string of one or more decimal digits. The number must end with a valid delimiter character. This function allows underscores in the interior of the number. The `stdin.geti32` function raises an appropriate exception if the input violates any of these rules or the value is outside the range plus or minus two billion. This function returns the binary form of the integer in the EAX register.

HLA high-level calling sequence examples:

```
stdin.geti32();
mov( eax, i32Var );
```

HLA low-level calling sequence examples:

```
call stdin.geti32;
mov( eax, i32Var );
```

**stdin.geti64(); @returns( "edx:eax" );**

This function reads a signed 64-bit decimal integer from the standard input device. The number may begin with any number of delimiter characters (see the `conv.setDelimiter` and `conv.getDelimiter` functions for details on the delimiter characters) followed by an optional minus sign and a string of one or more decimal digits. The number must end with a valid delimiter character. This function allows underscores in the interior of the number. The `stdin.geti64` function raises an appropriate exception if the input violates any of these rules or the value is outside the range of a 64-bit signed integer. This function returns the 64-bit result in EDX:EAX.

HLA high-level calling sequence examples:

```
stdin.geti64();
mov( edx, (type dword i64Var[4]) );
mov( eax, (type dword i64Var[0]) );
```

HLA low-level calling sequence examples:

```
call stdin.geti64;
mov( edx, (type dword i64Var[4]) );
mov( eax, (type dword i64Var[0]) );
```

```
stdin.geti128( var dest:lword );
```

This function reads a signed 128-bit decimal integer from the standard input. The number may begin with any number of delimiter characters (see the `conv.setDelimiter` and `conv.getDelimiter` functions for details on the delimiter characters) followed by an optional minus sign and a string of one or more decimal digits. The number must end with a valid delimiter character. This function allows underscores in the interior of the number. The `stdin.geti128` function raises an appropriate exception if the input violates any of these rules or the value is outside the range of a 128-bit signed integer. This function stores the 128-bit result in the `lword` you pass as a reference parameter.

HLA high-level calling sequence examples:

```
stdin.geti128( lwordVar );
```

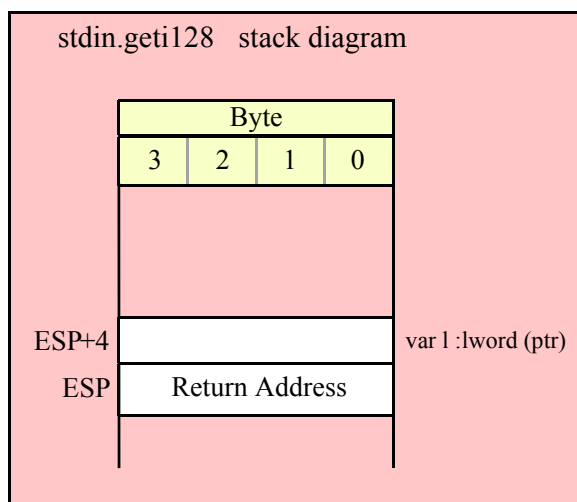
HLA low-level calling sequence examples:

```
// If lwordVar is a static variable:
```

```
pushd( &lwordVar );
call stdin.geti128;
```

```
// If lwordVar is a not static variable
// and a 32-bit register is available:
```

```
lea( eax, lwordVar ); // Assume EAX is available
push( eax );
call stdin.geti128;
```



## 30.8 Unsigned Integer Input Routines

```
stdin.getu8(); @returns( "al" );
```

This function reads an unsigned eight-bit decimal integer in the range 0..+255 from the standard input device. The number may begin with any number of delimiter characters (see the `conv.setDelimiter` and `conv.getDelimiter` functions for details on the delimiter characters) followed by a string of one or more decimal digits. The number must end with a valid delimiter character. This function allows underscores in the interior of the number. The `stdin.getu8` function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..255. This function returns the binary form of the integer in the AL register (zero extended into EAX).

HLA high-level calling sequence examples:

```
stdin.getu8();
mov( al, u8Var );
```

HLA low-level calling sequence examples:

```
call stdin.getu8;
mov( al, u8Var );
```

**stdin.getu16(); @returns( "ax" );**

This function reads an unsigned 16-bit decimal integer in the range 0..+65535 from the standard input device. The number may begin with any number of delimiter characters (see the `conv.setDelimiter` and `conv.getDelimiter` functions for details on the delimiter characters) followed by a string of one or more decimal digits. The number must end with a valid delimiter character. This function allows underscores in the interior of the number. The `stdin.getu16` function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..65535. This function returns the binary form of the integer in the AX register (zero extended into EAX).

HLA high-level calling sequence examples:

```
stdin.getu16();
mov( ax, u16Var );
```

HLA low-level calling sequence examples:

```
call stdin.getu16;
mov( ax, u16Var );
```

**stdin.getu32(); @returns( "eax" );**

This function reads an unsigned 32-bit decimal integer in the range 0..+4,294,967,295 from the standard input. The number may begin with any number of delimiter characters (see the `conv.setDelimiter` and `conv.getDelimiter` functions for details on the delimiter characters) followed by a string of one or more decimal digits. The number must end with a valid delimiter character. This function allows underscores in the interior of the number. The `stdin.getu32` function raises an appropriate exception if the input violates any of these rules or the value is outside the range 0..4,294,967,295. This function returns the binary form of the integer in the EAX register.

HLA high-level calling sequence examples:

```
stdin.getu32();
mov( eax, u32Var );
```

HLA low-level calling sequence examples:

```
call stdin.getu32;
mov( eax, u32Var );
```

```
stdin.getu64( ); @returns( "edx:eax" );
```

This function reads an unsigned 64-bit decimal integer from the standard input. The number may begin with any number of delimiter characters (see the `conv.setDelimiter` and `conv.getDelimiter` functions for details on the delimiter characters) followed by a string of one or more decimal digits. The number must end with a valid delimiter character or the end of the file. This function allows underscores in the interior of the number. The `stdin.getu64` function raises an appropriate exception if the input violates any of these rules or the value is outside the range  $0..2^{64}-1$ . This function returns the binary form of the integer in the the EDX:EAX register pair (EDX holds the H.O. dword).

HLA high-level calling sequence examples:

```
stdin.getu64();
mov( edx, (type dword u64Var[4]) );
mov( eax, (type dword u64Var[0]) );
```

HLA low-level calling sequence examples:

```
call stdin.getu64;
mov( edx, (type dword u64Var[4]) );
mov( eax, (type dword u64Var[0]) );
```

```
stdin.getu128( var dest:lword );
```

This function reads an unsigned 128-bit decimal integer from the standard input. The number may begin with any number of delimiter characters (see the `conv.setDelimiter` and `conv.getDelimiter` functions for details on the delimiter characters) followed by a string of one or more decimal digits. The number must end with a valid delimiter character. This function allows underscores in the interior of the number. The `stdin.getu128` function raises an appropriate exception if the input violates any of these rules or the value is outside the range  $0..2^{128}-1$ . This function returns the binary form of the integer in the `lword` parameter you pass by reference.

HLA high-level calling sequence examples:

```
stdin.getu128( lwordVar );
```

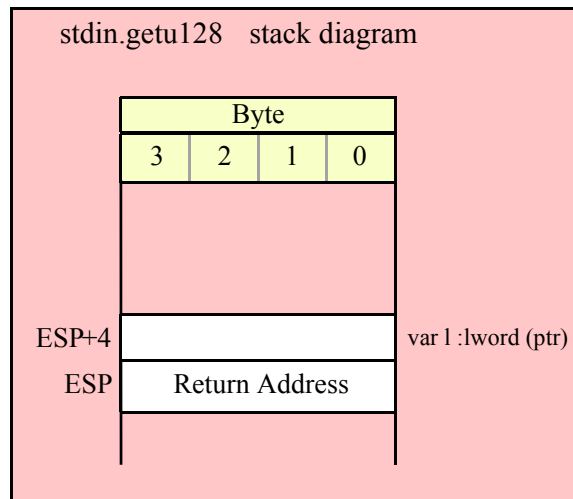
HLA low-level calling sequence examples:

```
// If lwordVar is a static variable:
```

```
pushd( &lwordVar );
call stdin.getu128;
```

```
// If lwordVar is a not static variable
// and a 32-bit register is available:
```

```
lea( eax, lwordVar ); // Assume EAX is available
push( eax );
call stdin.getu128;
```



## 30.9 Floating Point Input

**`stdin.getf();`**

This function reads an 80-bit floating point value in either decimal or scientific from the standard input and leaves the result sitting on the FPU stack. The number may begin with any number of delimiter characters (see the `conv.setDelimiter` and `conv.getDelimiter` functions for details on the delimiter characters) followed by an optional minus sign and a sequence of characters that represent a floating point value. The number must end with a valid delimiter character. This function allows underscores in the interior of the number. This function raises an appropriate exception if an error occurs.

HLA high-level calling sequence examples:

```
stdin.getf();
fstp( fpVar );
```

HLA low-level calling sequence examples:

```
call stdin.getf;
fstp( fpVar );
```

## 30.10 Generic File Input

**`stdin.get( List_of_items_to_read );`**

This is a macro that allows you to specify a list of variable names as parameters. The `stdin.get` macro reads an input value for each item in the list and stores the resulting value in each corresponding variable. This macro determines the type of each variable that you pass it and emits a call to the appropriate `stdin.getxxx` function to read the actual value. As an example, consider the following call to `filevar.get`:

```
stdin.get( i32, charVar, u16, strVar );
```

The macro invocation above expands into the following:

```
push( eax );
stdin.geti32( i32 );
stdin.getc();
mov( al, charVar );
```

```
stdin.geti16();  
mov( ax, u16 );  
stdin.gets( strVar );  
pop( eax );
```

Notice that `stdin.get` preserves the value in the EAX and EDX registers even though various `stdin.getxxx` functions use these registers. Note that `stdin.get` automatically handles the case where you specify EAX as an input variable and writes the value to [esp] so that it properly modifies EAX upon completion of the macro expansion.

Note that `stdin.get` supports eight-bit, 16-bit, 32-bit, 64-bit, and 128-bit input values. It automatically selects the appropriate input routine based on the type of the variable you specify.



## 31 The Standard Output Module (stdout.hhf)

This unit contains routines that write data to the standard output device. This is usually the console device, although the user may redirect the standard output to a file from the command line.

Note: be sure to read the chapter on "Passing Parameters to Standard Library Routines" (parmpassing.rtf) before reading this chapter.

**Note about stack diagrams:** this documentation includes stack diagrams for those functions that pass parameters on the stack. To conserve space, this documentation does not include a stack diagram for any function that does not pass data on the stack (that is, only a return address appears on the stack).

**A Note About Thread Safety:** Because the standard output device is a single resource, you will get inconsistent results if multiple threads attempt to write to the standard output device simultaneously. The HLA standard library stdout module does not attempt to synchronize thread access to the standard output device. If you are going to be writing to the standard output from multiple threads, it is your responsibility to ensure that the threads use properly synchronized access to this resource.

**A Note About the FPU:** The Standard Library code makes occasional use of the FPU, particularly when converting between real and string formats and when computing certain mathematical functions. You should exercise caution when using MMX instructions in a program that makes use of the Standard Library. In particular, you should ensure that you are always in FPU mode (by executing an EMMS instruction) after you are finished using MMX instructions. Better yet, you should avoid the MMX instruction set altogether and use the improved SSE instruction set that accomplishes the same tasks (and doesn't disturb the FPU).

### 31.1 Conversion Format Control

The standard output functions that convert numeric values to hexadecimal, unsigned decimal, and signed decimal output provide the ability to inject underscores between groups of three (decimal) or four (hexadecimal) digits to make it easier to read large numbers. You enable and disable underscore output using the `conv.setUnderscores` and `conv.getUnderscores` functions. Please refer to their documentation in the `conv.rtf` file for more details.

When converting numeric values to string form for output, the standard output routines call the conversion functions found in the `conv` (conversions) module. For detailed information on the actual conversions, please consult the `conv.rtf` document.

### 31.2 File I/O Routines and the Standard Output Handle

The standard output routines are basically a thin layer over the top of the fileio routines (see the fileio documentation for a complete description of those routines). Indeed, if you obtain the standard output handle, you can write data to the standard output device by passing this handle to a fileio function. Because the fileio module provides a slightly richer set of routines, there are a few instances where you might want to do this. You might also want to write a generic output function that expects a file handle and then pass it the standard output device file handle so that the function writes its output to the console (or other standard output device) rather than to some file. In any case, just be aware that it is perfectly reasonable to call fileio functions to write data to the standard output device.

```
stdout.handle; @returns( "eax" );
```

This routine returns the Linux/Windows handle for the Standard Output Device in the EAX register. You may use this handle with the file I/O routines to write data to the standard output device.

### 31.3 Standard Output Routines

The output routines in the stdout module are very similar to the file output routines in the stdout module. In general, these routines require (at least) one parameter: the value to write to the standard output. Some functions contain additional parameters that provide formatting information.

## 31.4 Miscellaneous Output Routines

```
stdout.write( var buffer:var; count:uns32 );
```

This procedure writes the number of bytes specified by the count variable to the standard output device. The bytes starting at the address of the buffer variable are written to the standard out. No range checking is done on the buffer, it is your responsibility to ensure that the buffer contains at least count valid data bytes. Because the buffer parameter is passed by untyped reference, a high-level style call to this function will take the address of whatever object you supply as the buffer parameter. *This includes pointer variables* (which is probably not what you want to do). Use the VAL keyword in a high-level style call if you want to use the value of a pointer variable rather than the address of that pointer variable (see the examples that follow).

HLA high-level calling sequence examples:

```
stdout.write( buffer, count );
```

```
// If "bufPtr" is dword containing the address of the buffer, then
// use the following code:
```

```
stdout.write( val bufPtr, bufferSize );
```

```
// If you actually want to write out the four bytes held by
// bufPtr (an unusual thing to do), you would use the
// following code:
```

```
stdout.write( bufPtr, 4 );
```

HLA low-level calling sequence examples:

```
// Assumes buffer is a static object at a fixed
// address in memory:
```

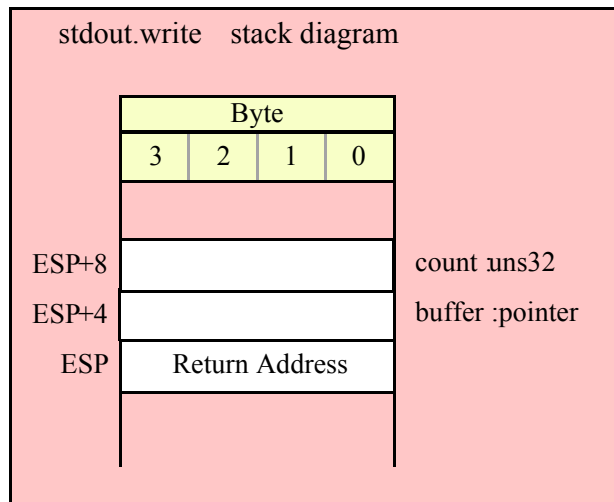
```
pushd( &buffer );
push( count );
call stdout.write;
```

```
// If a 32-bit register is available and buffer
// isn't at a fixed, static, address:
```

```
lea( eax, buffer );
push( eax );
push( count );
call stdout.write;
```

```
// If a 32-bit register is not available and buffer
// isn't at a fixed, static, address:
```

```
sub( 4, esp );
push( eax );
lea( eax, buffer );
mov( eax, [esp+4] );
pop( eax );
push( count );
call stdout.write;
```

**stdout.newln()**

This function writes a newline sequence (e.g., carriage return/line feed under Windows or line feed under Linux) to the output.

HLA high-level calling sequence examples:

```
stdout.newln();
```

HLA low-level calling sequence examples:

```
call stdout.newln;
```

## 31.5 Boolean Output

**stdout.putbool( b:boolean );**

This procedure writes the string "true" or "false" to the standard output depending on the value of the `b` parameter.

HLA high-level calling sequence examples:

```
stdout.putbool( boolVar );
```

```
// If the boolean is in a register (AL):
```

```
stdout.putbool( al );
```

HLA low-level calling sequence examples:

```
// If "boolVar" is not one of the last three
// bytes on a page of memory, you can do this:
```

```

push( (type dword boolVar ) );
call stdout.putbool;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( boolVar , eax ); // Assume EAX is available
push( eax );
call stdout.putbool;

// If no register is available, do something
// like the following code:

sub( 4, esp );
push( eax );
movzx( boolVar , eax );
mov( eax, [esp+4] );
pop( eax );
call stdout.putbool;

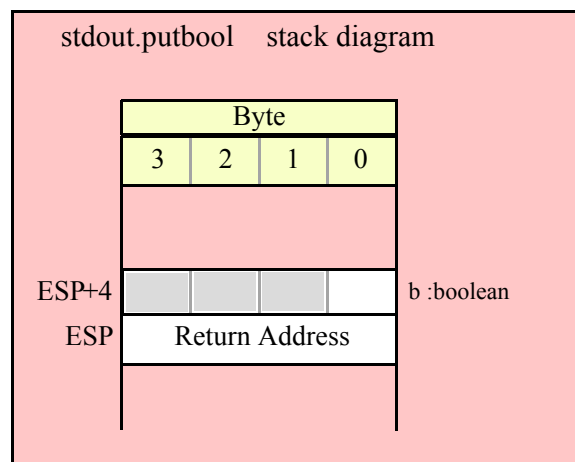
// If the boolean value is in al, bl, cl, or dl
// then you can use code like the following:

push( eax ); // Assume boolVar is in AL
call stdout.putbool;

// If the Boolean value is in ah, bh, ch, or dh
// you'll have to use code like the following:

xchg( al, ah ); // Assume boolVar is in AH
push( eax );    // It's now in AL
xchg( al, ah ); // Restore al/ah
call stdout.putbool;

```



## 31.6 Character, String, and Character Set Output Routines

```
stdout.putc( c:char );
```

Writes the character specified by the c parameter to the standard output device.

HLA high-level calling sequence examples:

```
stdout.putc( charVar );

// If the character is in a register (AL):

stdout.putc( al );
```

HLA low-level calling sequence examples:

```
// If "charVar" is not one of the last three
// bytes on a page of memory, you can do this:

push( (type dword charVar) );
call stdout.putc;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( charVar, eax ); // Assume EAX is available
push( eax );
call stdout.putc;

// If no register is available, do something
// like the following code:

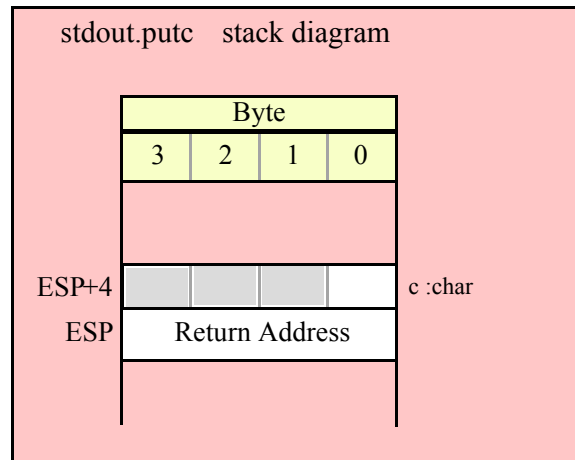
sub( 4, esp );
push( eax );
movzx( charVar, eax );
mov( eax, [esp+4] );
pop( eax );
call stdout.putc;

// If the character value is in al, bl, cl, or dl
// then you can use code like the following:

push( eax ); // Assume charVar is in AL
call stdout.putc;

// If the character value is in ah, bh, ch, or dh
// you'll have to use code like the following:

xchg( al, ah ); // Assume charVar is in AH
push( eax );    // It's now in AL
xchg( al, ah ); // Restore al/ah
call stdout.putc;
```



```
stdout.putcSize( c:char; width:int32; fill:char )
```

Outputs the character `c` to the standard output using at least `width` output positions. If the absolute value of `width` is greater than one, then this function writes `fill` characters as padding characters during the output. If `width` is a positive value greater than one, then `stdout.putcSize` writes `c` left justified in a field of `width` characters; if `width` is a negative value less than one, then `stdout.putcSize` writes `c` right justified in a field of `width` characters.

HLA high-level calling sequence examples:

```
stdout.putcSize( charVar, width, padChar );
```

HLA low-level calling sequence examples:

```
// If "charVar" and "padChar" are not one of the last three
// bytes on a page of memory, you can do this:

push( (type dword charVar) );
push( width );
push( (type dword padChar) );
call stdout.putcSize;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( charVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call stdout.putcSize;

// If no registers are available, do something
// like the following code:

push( eax );
movzx( charVar, eax );
push( eax );
push( width );
```

```

movzx( padChar, eax );
push( eax );
call stdout.putcSize;
pop( eax );

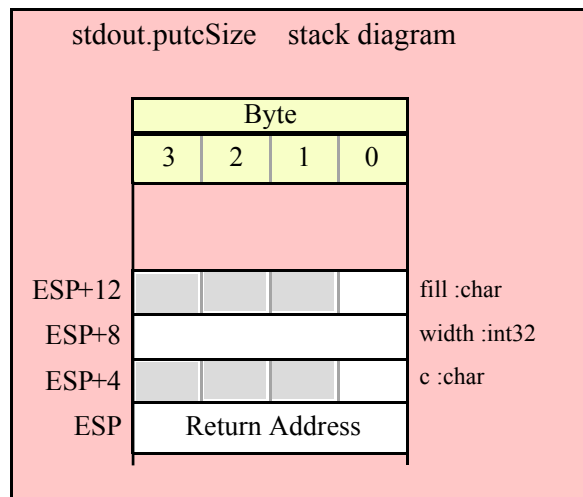
// If "charVar" or "padChar" are in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:

push( eax );    // Assume charVar is in AL
push( width );
push( ebx );    // Assume padChar is in BL
call stdout.putcSize;

// Do the following if the characters are
// in AH, BH, CH, or DH:

xchg( al, ah );    // Assume charVar is in AH
xchg( bl, bh );    // Assume padChar is in BH
push( eax );
push( width );
push( ebx );
xchg( al, ah );
xchg( bl, bh );
call stdout.putcSize;

```



### **stdout.putcset( cst:cset );**

This function writes all the members of the `cst` character set parameter to the standard output device.

HLA high-level calling sequence examples:

```

stdout.putcset( csVar );
stdout.putcset( [ebx] ); // EBX points at the cset.

```

HLA low-level calling sequence examples:

```

push( (type dword csVar[12]) ); // Push H.O. dword first
push( (type dword csVar[8]) );
push( (type dword csVar[4]) );

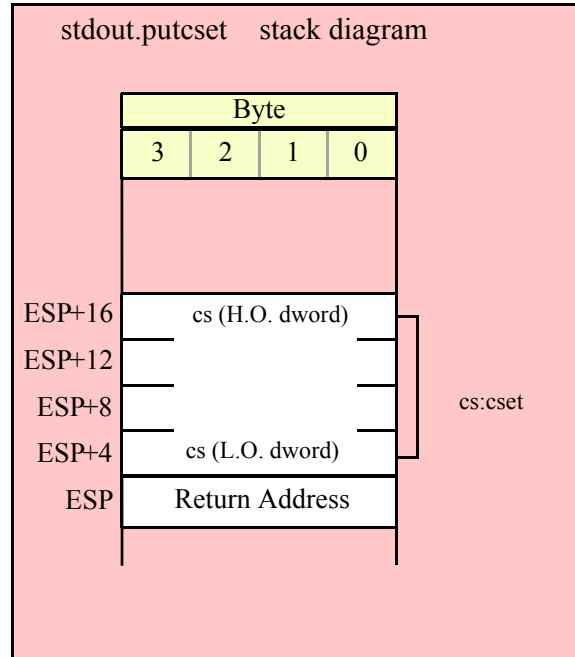
```

```

push( (type dword csVar) );      // Push L.O. dword last
call stdout.putcset;

push( (type dword [ebx+12]) );   // Push H.O. dword first
push( (type dword [ebx+8]) );
push( (type dword [ebx+4]) );
push( (type dword [ebx]) );      // Push L.O. dword last
call stdout.putcset;

```



**stdout.puts( s:string );**

This procedure writes the value of the string parameter to the standard output. Remember, string values are actually 4-byte pointers to the string's character data.

HLA high-level calling sequence examples:

```

stdout.puts( strVar );
stdout.puts( ebx ); // EBX holds a string value.
stdout.puts( "Hello World" );

```

HLA low-level calling sequence examples:

// For string variables:

```

push( strVar );
call stdout.puts;

```

// For string values held in registers:

```

push( ebx ); // Assume EBX holds the string value
call stdout.puts;

```

```

// For string literals, assuming a 32-bit register
// is available:

```



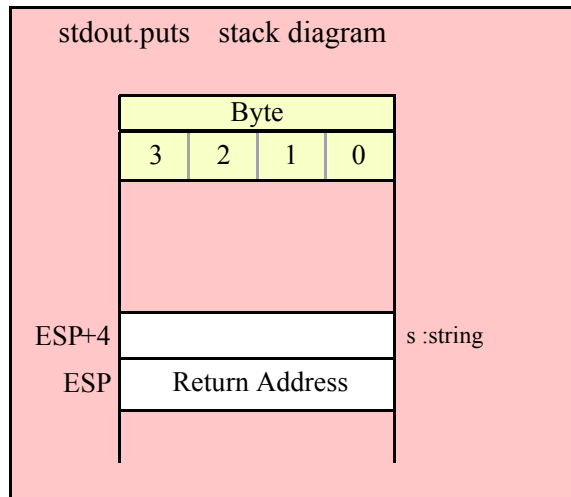
```

lea( eax, "Hello World" ); // Assume EAX is available.
push( eax );
call stdout.puts;

// If a 32-bit register is not available:

readonly
  literalString :string := "Hello World";
  .
  .
  .
push( literalString );
call stdout.puts;

```



```
stdout.putsSize( s:string; width:int32; fill:char );
```

This function writes the *s* string to the standard output using at least *width* character positions. If the absolute value of *width* is less than or equal to the length of *s*, then this function behaves exactly like `stdout.puts`. On the other hand, if the absolute value of *width* is greater than the length of *s*, then `stdout.putsSize` writes *width* characters to the standard output. This procedure emits the fill character in the extra print positions. If *width* is positive, then `stdout.putsSize` right justifies the string in the print field. If *width* is negative, then `stdout.putsSize` left justifies the string in the print field. Generally, people expect the string to be left justified, so you should ensure that this value is negative to achieve this.

HLA high-level calling sequence examples:

```

stdout.putsSize( strVar, width, ' ' );

// For the following, EBX holds the string value,
// ECX contains the width, and AL holds the pad
// character:

stdout.putsSize( ebx, ecx, al );

stdout.putsSize( "Hello World", 25, padChar );

```

HLA low-level calling sequence examples:

```
// For string variables:
```

```

push( strVar );
push( width );
pushd( ' ' );
call stdout.putsSize;

// For string values held in registers:

push( ebx ); // Assume EBX holds the string value
push( ecx ); // Assume ECX holds the width
push( eax ); // Assume AL holds the fill character
call stdout.putsSize;

// For string literals, assuming a 32-bit register
// is available:

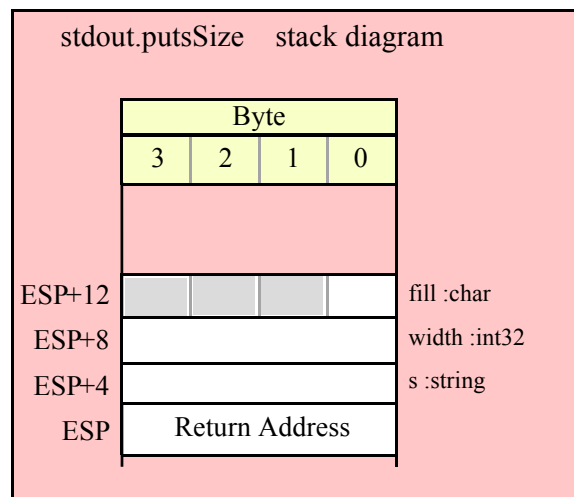
lea( eax, "Hello World" ); // Assume EAX is available.
push( eax );
pushd( 25 );
movzx( padChar, eax );
push( eax );
call stdout.putsSize;

// If a 32-bit register is not available:

readonly
literalString :string := "Hello World";

// Note: element zero is the actual pad character.
// The other elements are just padding.
padChar :char[4] := [ '.', #0, #0, #0 ];
.
.
.
push( literalString );
pushd( 25 );
push( (type dword padChar) );
call stdout.putsSize;

```



## 31.7 Hexadecimal Output Routines

These routines convert numeric data to hexadecimal string form (using the hexadecimal conversion routines found in the `conv` module) and write the resulting string to the standard output device.

**stdout.putb( b:byte )**

This procedure writes the value of `b` to the standard output using exactly two hexadecimal digits (including a leading zero if necessary).

HLA high-level calling sequence examples:

```
stdout.putb( byteVar );

// If the character is in a register (AL):

stdout.putb( al );
```

HLA low-level calling sequence examples:

```
// If "byteVar " is not one of the last three
// bytes on a page of memory, you can do this:

push( (type dword byteVar ) );
call stdout.putb;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( byteVar , eax ); // Assume EAX is available
push( eax );
call stdout.putb;

// If no register is available, do something
// like the following code:

sub( 4, esp );
push( eax );
movzx( byteVar , eax );
mov( eax, [esp+4] );
pop( eax );
call stdout.putb;

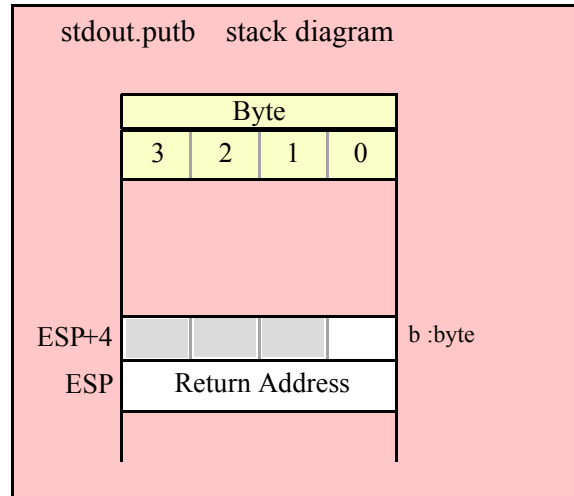
// If the character value is in al, bl, cl, or dl
// then you can use code like the following:

push( eax ); // Assume byteVar is in AL
call stdout.putb;

// If the byte value is in ah, bh, ch, or dh
// you'll have to use code like the following:

xchg( al, ah ); // Assume byteVar is in AH
push( eax );    // It's now in AL
xchg( al, ah ); // Restore al/ah
```

```
call stdout.putb;
```



```
stdout.puth8( b:byte );
```

This procedure writes the value of `b` to the standard output using the minimum necessary number of hexadecimal digits.

HLA high-level calling sequence examples:

```
stdout.puth8( byteVar );
```

```
// If the character is in a register (AL):
```

```
stdout.puth8( al );
```

HLA low-level calling sequence examples:

```
// If "byteVar " is not one of the last three
// bytes on a page of memory, you can do this:
```

```
push( (type dword byteVar ) );
call stdout.puth8;
```

```
// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:
```

```
movzx( byteVar , eax ); // Assume EAX is available
push( eax );
call stdout.puth8;
```

```
// If no register is available, do something
// like the following code:
```

```
sub( 4, esp );
push( eax );
```

```

movzx( byteVar , eax );
mov( eax, [esp+4] );
pop( eax );
call stdout.puth8;

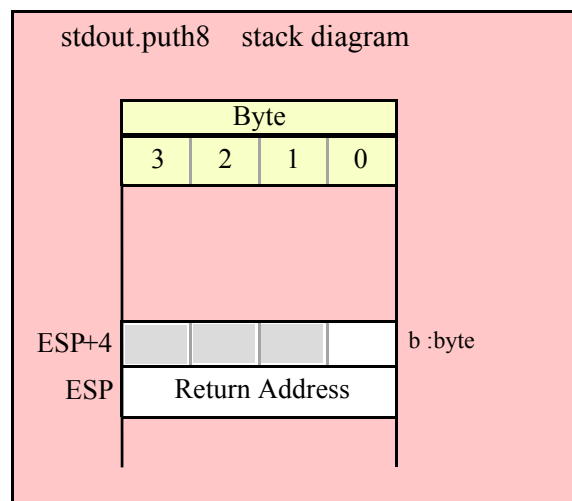
// If the character value is in al, bl, cl, or dl
// then you can use code like the following:

push( eax ); // Assume byteVar is in AL
call stdout.puth8;

// If the byte value is in ah, bh, ch, or dh
// you'll have to use code like the following:

xchg( al, ah ); // Assume byteVar is in AH
push( eax );    // It's now in AL
xchg( al, ah ); // Restore al/ah
call stdout.puth8;

```



### **stdout.puth8Size( b:byte; size:dword; fill:char )**

The `stdout.puth8Size` function writes an 8-bit hexadecimal value to the standard output allowing you specify a minimum field width and a fill character.

HLA high-level calling sequence examples:

```
stdout.puth8Size( byteVar, width, padChar );
```

HLA low-level calling sequence examples:

```

// If "byteVar" and "padChar" are not one of the last three
// bytes on a page of memory, you can do this:

push( (type dword byteVar) );
push( width );
push( (type dword padChar) );
call stdout.puth8Size;

// If you can't guarantee that the previous code

```

```

// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( byteVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call stdout.puth8Size;

// If no registers are available, do something
// like the following code:

push( eax );
movzx( byteVar, eax );
push( eax );
push( width );
movzx( padChar, eax );
push( eax );
call stdout.puth8Size;
pop( eax );

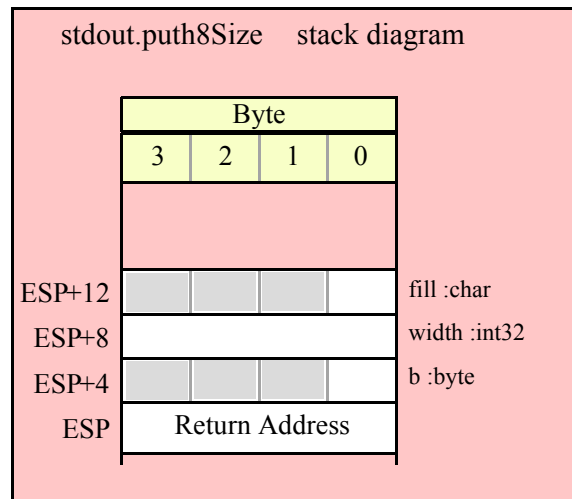
// If "byteVar" or "padChar" are in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:

push( eax ); // Assume byteVar is in AL
push( width );
push( ebx ); // Assume padChar is in BL
call stdout.puth8Size;

// Do the following if the characters are
// in AH, BH, CH, or DH:

xchg( al, ah ); // Assume byteVar is in AH
xchg( bl, bh ); // Assume padChar is in BH
push( eax );
push( width );
push( ebx );
xchg( al, ah );
xchg( bl, bh );
call stdout.puth8Size;

```



```
stdout.putw( w:word )
```

This procedure writes the value of w to the standard output device using exactly four hexadecimal digits (including leading zeros if necessary).

HLA high-level calling sequence examples:

```
stdout.putw( wordVar );
```

```
// If the word is in a register (AX):
```

```
stdout.putw( ax );
```

HLA low-level calling sequence examples:

```
// If "wordVar " is not one of the last three
// bytes on a page of memory, you can do this:
```

```
push( (type dword wordVar) );
call stdout.putw;
```

```
// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:
```

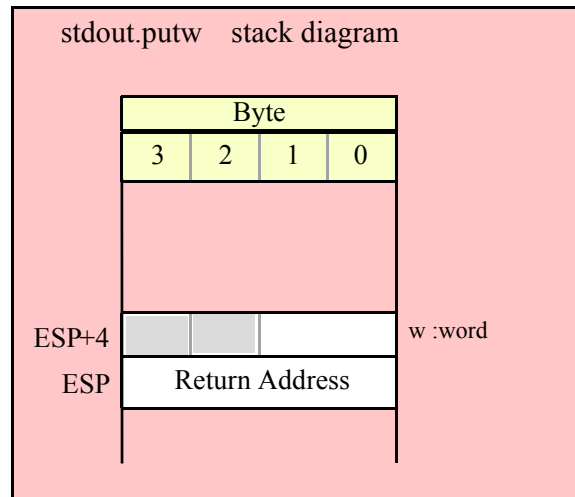
```
movzx( wordVar, eax ); // Assume EAX is available
push( eax );
call stdout.putw;
```

```
// If no register is available, do something
// like the following code:
```

```
push( eax );
movzx( wordVar, eax );
push( eax );
call stdout.putw;
pop( eax );
```

```
// If the word value is in a 16-bit register
// then you can use code like the following:

push( eax ); // Assume wordVar is in AX
call stdout.putw;
```



**stdout.puth16( w:word )**

This procedure writes the value of w to the standard out using the minimum necessary number of hexadecimal digits.

HLA high-level calling sequence examples:

```
stdout.puth16( wordVar );

// If the word is in a register (AX):

stdout.puth16( ax );
```

HLA low-level calling sequence examples:

```
// If "wordVar " is not one of the last three
// bytes on a page of memory, you can do this:

push( (type dword wordVar) );
call stdout.puth16;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( wordVar, eax ); // Assume EAX is available
push( eax );
call stdout.puth16;

// If no register is available, do something
```

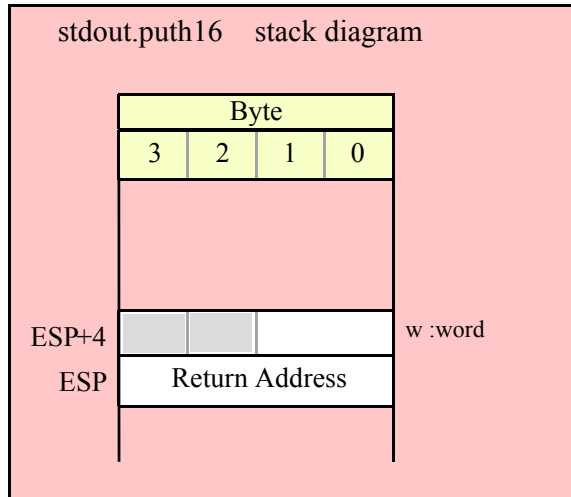


```
// like the following code:

push( eax );
movzx( wordVar, eax );
push( eax );
call stdout.puth16;
pop( eax );

// If the word value is in a 16-bit register
// then you can use code like the following:

push( eax ); // Assume wordVar is in AX
call stdout.puth16;
```



**stdout.puth16Size( w:word; size:dword; fill:char )**

The `stdout.puth16Size` function writes a 16-bit hexadecimal value to the standard output allowing you specify a minimum field width and a fill character.

HLA high-level calling sequence examples:

```
stdout.puth16Size( wordVar, width, padChar );
```

HLA low-level calling sequence examples:

```
// If "wordVar" and "padChar" are not one of the last three
// bytes on a page of memory, you can do this:

push( (type dword wordVar) );
push( width );
push( (type dword padChar) );
call stdout.puth16Size;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( wordVar, eax ); // Assume EAX is available
push( eax );
```

```

push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call stdout.puth16Size;

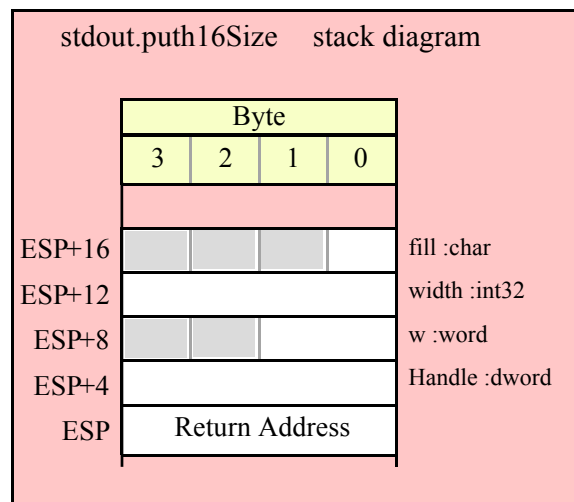
// If no registers are available, do something
// like the following code:

push( eax );
movzx( wordVar, eax );
push( eax );
push( width );
movzx( padChar, eax );
push( eax );
call stdout.puth16Size;
pop( eax );

// If "wordVar" is in a 16-bit register
// and "padChar" is in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:

push( eax ); // Assume wordVar is in AX
push( width );
push( ebx ); // Assume padChar is in BL
call stdout.puth16Size;

```



### **stdout.putd( d:dword )**

This procedure writes the value of *d* to the standard out using exactly eight hexadecimal digits (including leading zeros if necessary), if underscore output is not enabled. This routine will emit nine characters (eight digits plus an underscore) if underscore output is enabled.

HLA high-level calling sequence examples:

```

stdout.putd( dwordVar );

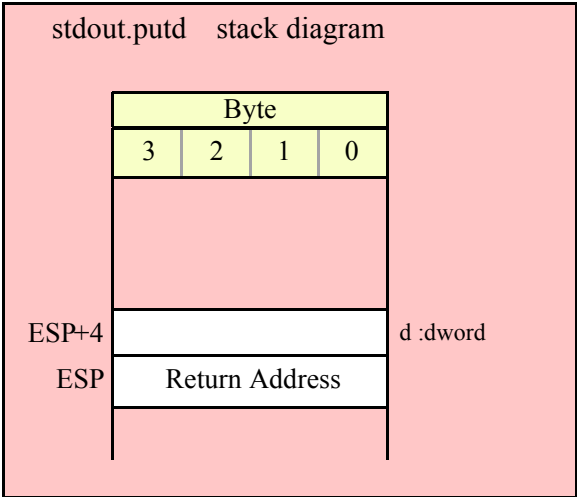
// If the dword value is in a register (EAX):

```

```
stdout.putd( eax );
```

HLA low-level calling sequence examples:

```
push( dwordVar );  
call stdout.putd;  
  
push( eax );  
call stdout.putd;
```



```
stdout.puth32( d:dword );
```

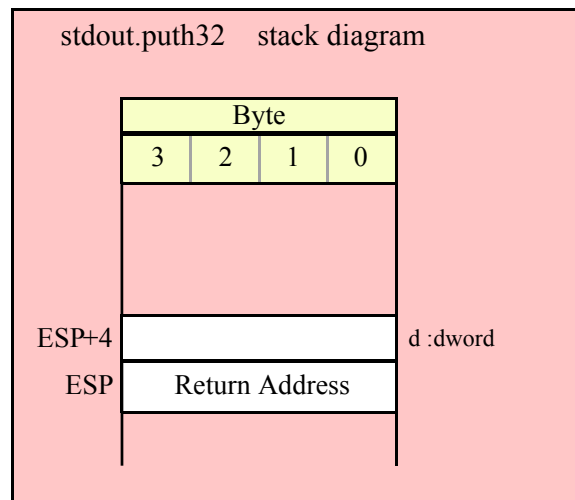
This procedure writes the value of `d` to the standard output using the minimum number of hexadecimal digits necessary. If underscore output is enabled (see `conv.setUnderscores` and `conv.getUnderscores`) then this function will emit an underscore between groups of four hexadecimal digits, starting from the least significant digit.

HLA high-level calling sequence examples:

```
stdout.puth32( dwordVar );  
  
// If the dword is in a register (EAX):  
  
stdout.puth32( eax );
```

HLA low-level calling sequence examples:

```
push( dwordVar );  
call stdout.puth32;  
  
push( eax );  
call stdout.puth32;
```



**stdout.puth32Size( d:dword; size:dword; fill:char )**

The stdout.puth32Size function outputs d as a hexadecimal string (including underscores, if enabled) and it allows you specify a minimum field width and a fill character.

HLA high-level calling sequence examples:

```
stdout.puth32Size( dwordVar, width, ' ' );

// If the dword is in a register (EAX):

stdout.puth32Size( eax, width, cl );
```

HLA low-level calling sequence examples:

```
push( dwordVar );
push( width );
pushd( ' ' );
call stdout.puth32Size;

push( eax );
push( width );
push( ecx ); // fill char is in CL
call stdout.puth32Size;

// Assume fill char is in CH

push( eax );
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call stdout.puth32Size;

// Alternate method of the above

push( eax );
push( width );
```

```
sub( 4, esp );
mov( ch, [esp] );
call stdout.puth32Size;

// If the fill char is a variable and
// a register is available, try this code:

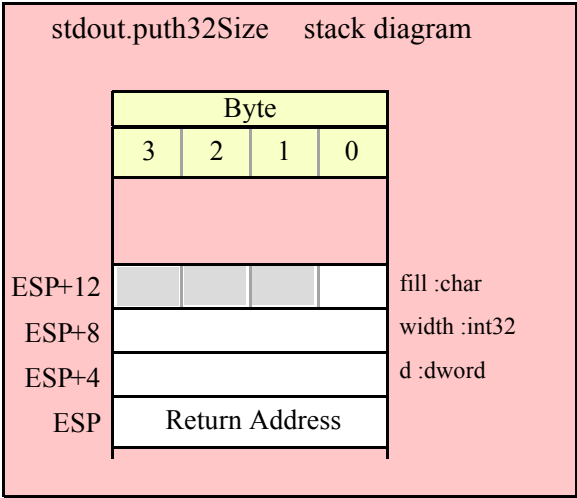
push( eax );
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call stdout.puth32Size;

// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( eax );
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call stdout.puth32Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( eax );
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call stdout.puth32Size;
```



```
stdout.putq( q:qword );
```

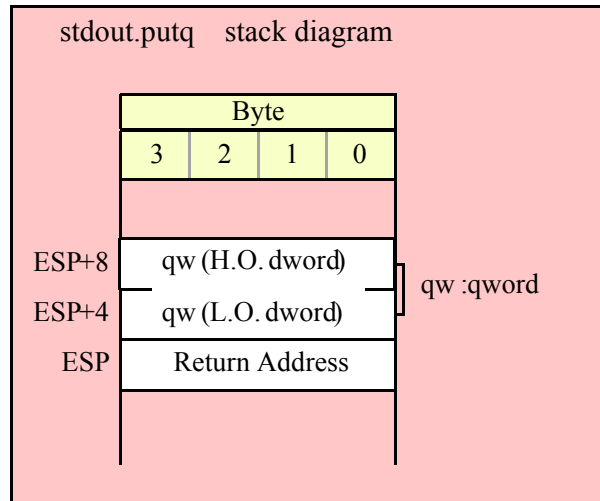
This procedure writes the value of q to the standard output device using exactly sixteen hexadecimal digits (including leading zeros if necessary and an intervening underscores if underscore output is enabled).

HLA high-level calling sequence examples:

```
stdout.putq( qwordVar );
```

HLA low-level calling sequence examples:

```
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
call stdout.putq;
```



```
stdout.puth64( q:qword );
```

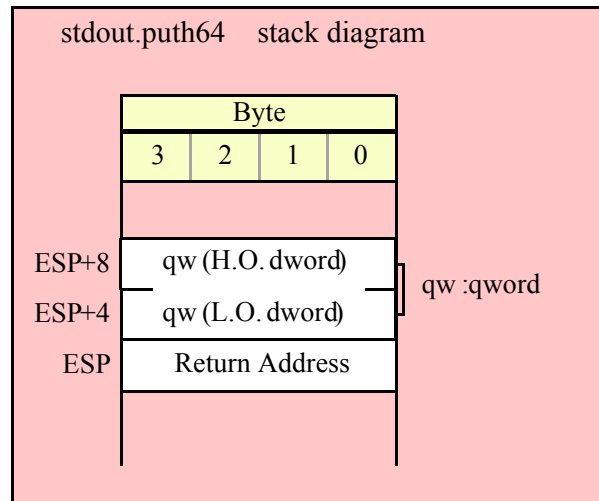
This procedure writes the value of q to the standard output using the minimum necessary number of hexadecimal digits (including intervening underscores if underscore output is enabled).

HLA high-level calling sequence examples:

```
stdout.puth64( qwordVar );
```

HLA low-level calling sequence examples:

```
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
call stdout.puth64;
```



```
stdout.puth64Size( q:qword; size:dword; fill:char );
```

The `stdout.putqSize` function lets you specify a minimum field width and a fill character. The `stdout.putq` routine uses a minimum size of two and a fill character of '0'. Note that if underscore output is enabled, this routine will emit 19 characters (16 digits plus three underscores).

HLA high-level calling sequence examples:

```
stdout.puth64Size( qwordVar, width, ' ' );
```

HLA low-level calling sequence examples:

```
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
pushd( ' ' );
call stdout.puth64Size;
```

```
push( edx ); // Assume 64-bit value in edx:eax
push( eax );
push( width );
push( ecx ); // fill char is in CL
call stdout.puth64Size;
```

```
// Assume fill char is in CH
```

```
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call stdout.puth64Size;
```

```
// Alternate method of the above
```

```

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));      // L.O. dword last
push( width );
sub( 4, esp );
mov( ch, [esp] );
call stdout.puth64Size;

// If the fill char is a variable and
// a register is available, try this code:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));      // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call stdout.puth64Size;

// If the fill char is a variable and
// no register is available, here's one
// possibility:

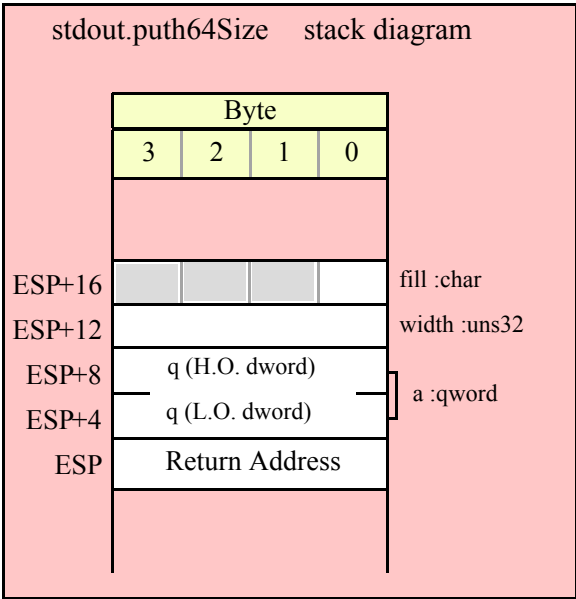
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));      // L.O. dword last
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call stdout.puth64Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));      // L.O. dword last
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call stdout.puth64Size;

```





**stdout.puttb( tb:tbyte );**

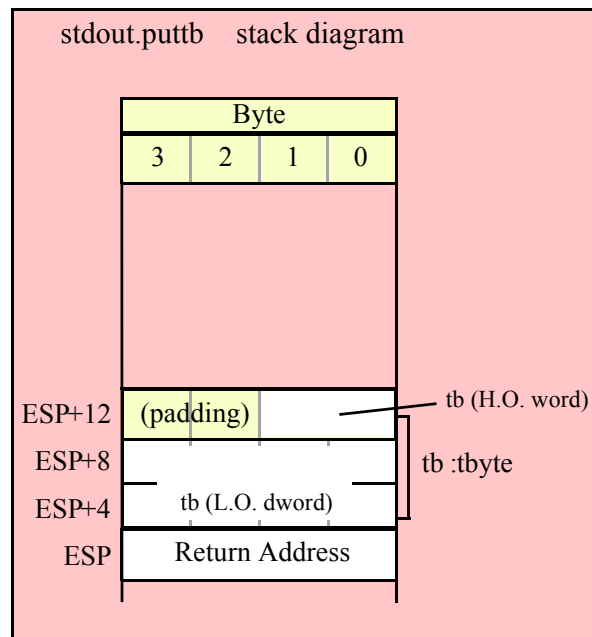
This procedure writes the value of tb to the standard out using exactly 20 hexadecimal digits (including leading zeros if necessary and an intervening underscores if underscore output is enabled).

HLA high-level calling sequence examples:

```
stdout.puttb( tbyteVar );
```

HLA low-level calling sequence examples:

```
pushw( 0 );           // Push push a 0 pad word
push( (type word tbyteVar[8])); // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));    // L.O. dword last
call stdout.puttb;
```



```
stdout.puth80( tb:tbyte );
```

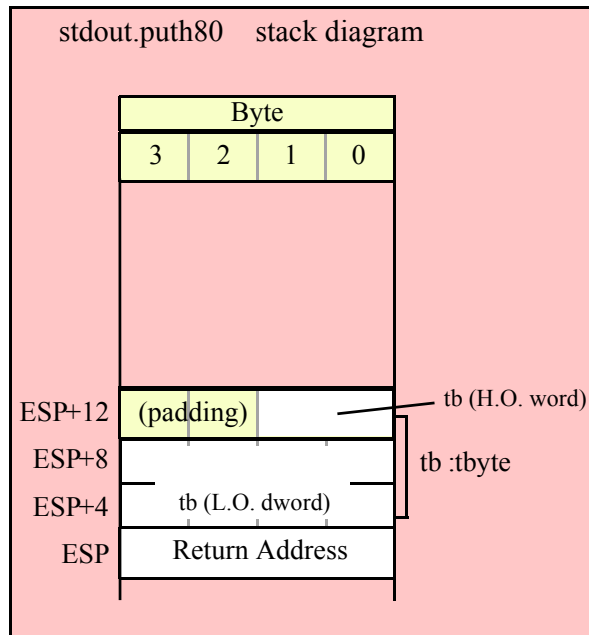
This procedure writes the value of tb to the standard output using the minimum necessary number of hexadecimal digits (including intervening underscores if underscore output is enabled).

HLA high-level calling sequence examples:

```
stdout.puth80( tbyteVar );
```

HLA low-level calling sequence examples:

```
pushw( 0 );           // Push push a 0 pad word
push( (type word tbyteVar[8])); // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));    // L.O. dword last
call stdout.puth80;
```



```
stdout.puth80Size( tb:tbyte; size:dword; fill:char );
```

The `stdout.puth80Size` function lets you specify a minimum field width and a fill character. It writes the `tbyte` value `tb` as a hexadecimal string to the standard output device using the provided minimum size and fill character.

HLA high-level calling sequence examples:

```
stdout.puth80Size( tbyteVar, width, ' ' );
```

HLA low-level calling sequence examples:

```
pushw( 0 );           // Push push a 0 pad word
push( (type word tbyteVar[8])); // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));    // L.O. dword last
push( width );
pushd( ' ' );
call stdout.puth80Size;
```

```
// Assume fill char is in CH
```

```
pushw( 0 );           // Push push a 0 pad word
push( (type word tbyteVar[8])); // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));    // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call stdout.puth80Size;
```

```

// Alternate method of the above

pushw( 0 );           // Push push a 0 pad word
push( (type word tbyteVar[8])); // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));    // L.O. dword last
push( width );
sub( 4, esp );
mov( ch, [esp] );
call stdout.puth80Size;

// If the fill char is a variable and
// a register is available, try this code:

pushw( 0 );           // Push push a 0 pad word
push( (type word tbyteVar[8])); // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));    // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call stdout.puth80Size;

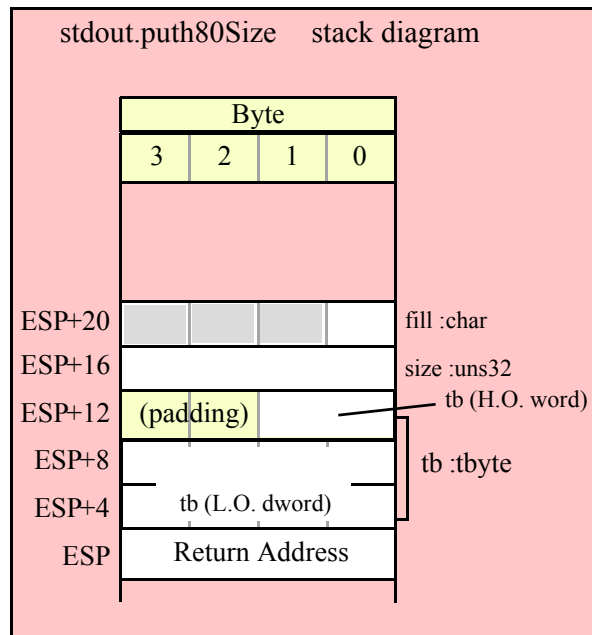
// If the fill char is a variable and
// no register is available, here's one
// possibility:

pushw( 0 );           // Push push a 0 pad word
push( (type word tbyteVar[8])); // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));    // L.O. dword last
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call stdout.puth80Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

pushw( 0 );           // Push push a 0 pad word
push( (type word tbyteVar[8])); // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));    // L.O. dword last
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call stdout.puth80Size;

```



```
stdout.putl( l:ldword );
```

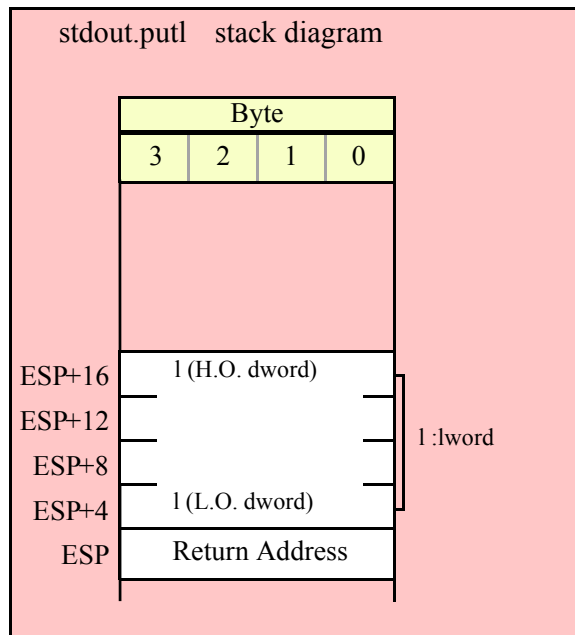
This procedure writes the value of `l` to the standard output using exactly 32 hexadecimal digits (including leading zeros if necessary and an intervening underscores if underscore output is enabled).

HLA high-level calling sequence examples:

```
stdout.putl( lwordVar );
```

HLA low-level calling sequence examples:

```
push( (type dword lwordVar[12]) ); // H.O. dword first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
call stdout.putl;
```



```
stdout.puth128( l:lword );
```

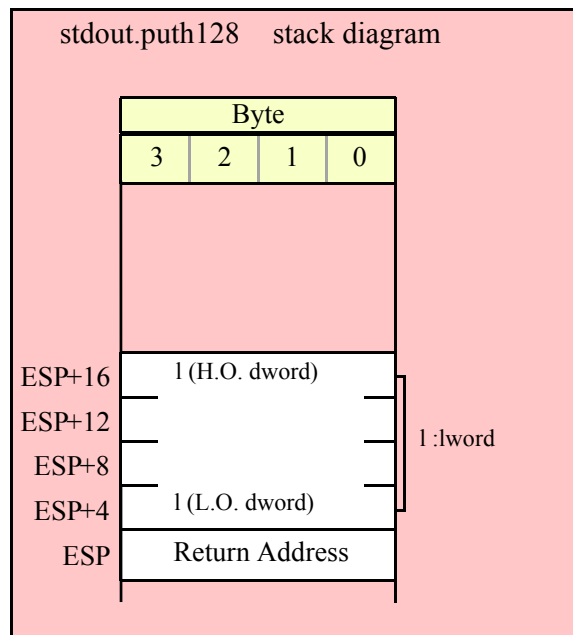
This procedure writes the value of `l` to the standard output using the minimum necessary number of hexadecimal digits (including intervening underscores if underscore output is enabled).

HLA high-level calling sequence examples:

```
stdout.puth128( lwordVar );
```

HLA low-level calling sequence examples:

```
push( (type dword lwordVar[12]) ); // H.O. dword first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar) );    // L.O. dword last
call stdout.puth128;
```



```
stdout.puth128Size( 1:lword; size:dword; fill:char );
```

The `stdout.puth128Size` function writes an `lword` value to the standard output and it lets you specify a minimum field width and a fill character.

HLA high-level calling sequence examples:

```
stdout.puth128Size( tbyteVar, width, ' ' );
```

HLA low-level calling sequence examples:

```
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar)); // L.O. dword last
push( width );
pushd( ' ' );
call stdout.puth128Size;
```

```
// Assume fill char is in CH
```

```
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar)); // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call stdout.puth128Size;
```

```

// Alternate method of the above

push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));    // L.O. dword last
push( width );
sub( 4, esp );
mov( ch, [esp] );
call stdout.puth128Size;

// If the fill char is a variable and
// a register is available, try this code:

push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));    // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call stdout.puth128Size;

// If the fill char is a variable and
// no register is available, here's one
// possibility:

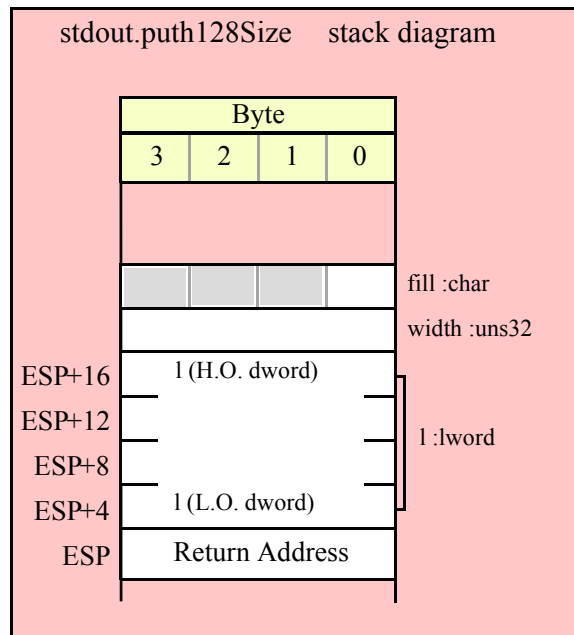
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));    // L.O. dword last
push( width );
push( (type dword fillChar) );   // Chance of page crossing!
call stdout.puth128Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));    // L.O. dword last
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call stdout.puth128Size;

```





## 31.8 Signed Integer Output Routines

These routines convert signed integer values to string format and write that string to the standard output device. The `stdout.putxxxSize` functions contain width and fill parameters that let you specify the minimum field width when outputting a value.

If the absolute value of width is greater than the number of print positions the value requires, then these functions output width characters to the standard output device. If width is non-negative, then these functions right-justify the value in the output field; if value is negative, then these functions left-justify the value in the output field.

These functions print the fill character as the padding value for the extra print positions.

Note that unlike floating point values, these functions do not print a space in front of the value if it is non-negative.

```
xxxSize( value, width, fill );
```

Assuming “value” requires five print positions, “width” is eight, and fill is “f” then the `xxxSize` functions produce the string

f	f	f	V	A	L	U	E
---	---	---	---	---	---	---	---

Assuming “value” requires five print positions, “width” is minus eight, and fill is “f” then the `xxxSize` functions produce the string

V	A	L	U	E	f	f	f
---	---	---	---	---	---	---	---

```
stdout.puti8 ( b:byte );
```

This function converts the eight-bit signed integer you pass as a parameter to a string and writes this string to the standard output using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
stdout.puti8( byteVar );
```

```
// If the character is in a register (AL):
```

```
stdout.puti8( al );
```

HLA low-level calling sequence examples:

```
// If "byteVar " is not one of the last three
// bytes on a page of memory, you can do this:

push( (type dword byteVar ) );
call stdout.puti8;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( byteVar , eax ); // Assume EAX is available
push( eax );
call stdout.puti8;

// If no register is available, do something
// like the following code:

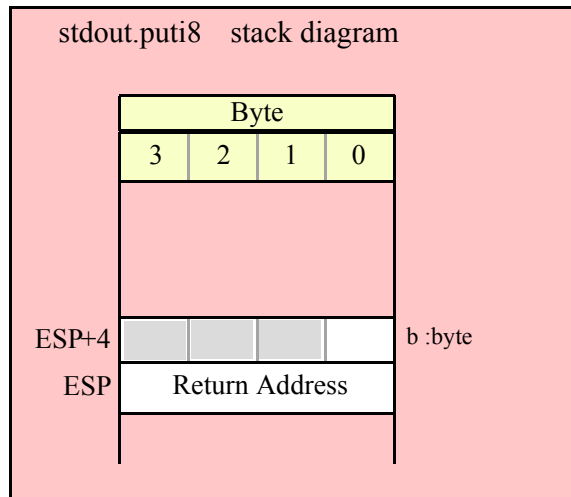
push( eax );
movzx( byteVar , eax );
push( eax );
call stdout.puti8;
pop( eax );

// If the character value is in al, bl, cl, or dl
// then you can use code like the following:

push( eax ); // Assume byteVar is in AL
call stdout.puti8;

// If the byte value is in ah, bh, ch, or dh
// you'll have to use code like the following:

xchg( al, ah ); // Assume byteVar is in AH
push( eax );    // It's now in AL
xchg( al, ah ); // Restore al/ah
call stdout.puti8;
```



**stdout.puti8Size ( b:byte; width:int32; fill:char )**

This function writes the eight-bit signed integer value you pass to the standard output using the width and fill values as specified above.

HLA high-level calling sequence examples:

```
stdout.puti8Size( byteVar, width, padChar );
```

HLA low-level calling sequence examples:

```
// If "byteVar" and "padChar" are not one of the last three
// bytes on a page of memory, you can do this:
```

```
push( (type dword byteVar) );
push( width );
push( (type dword padChar) );
call stdout.puti8Size;
```

```
// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:
```

```
movzx( byteVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call stdout.puti8Size;
```

```
// If no registers are available, do something
// like the following code:
```

```
push( eax );
movzx( byteVar, eax );
push( eax );
push( width );
movzx( padChar, eax );
push( eax );
```

```

call stdout.puti8Size;
pop( eax );

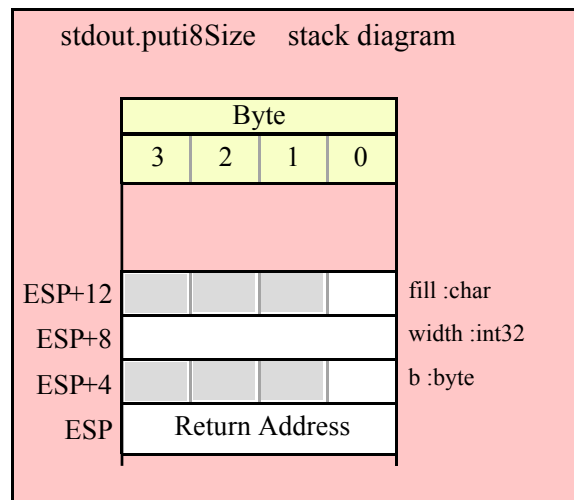
// If "byteVar" or "padChar" are in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:

push( eax );    // Assume byteVar is in AL
push( width );
push( ebx );    // Assume padChar is in BL
call stdout.puti8Size;

// Do the following if the characters are
// in AH, BH, CH, or DH:

xchg( al, ah );    // Assume byteVar is in AH
xchg( bl, bh );    // Assume padChar is in BH
push( eax );
push( width );
push( ebx );
xchg( al, ah );
xchg( bl, bh );
call stdout.puti8Size;

```



```
stdout.puti16( w:word );
```

This function converts the 16-bit signed integer you pass as a parameter to a string and writes this string to the standard output device using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```

stdout.puti16( wordVar );

// If the word is in a register (AX):

stdout.puti16( ax );

```

HLA low-level calling sequence examples:

```

// If "wordVar " is not one of the last three
// bytes on a page of memory, you can do this:

push( (type dword wordVar) );
call stdout.puti16;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( wordVar, eax ); // Assume EAX is available
push( eax );
call stdout.puti16;

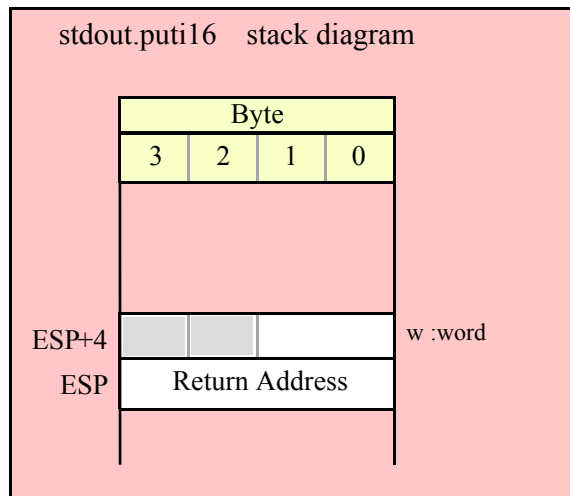
// If no register is available, do something
// like the following code:

push( eax );
movzx( wordVar, eax );
push( eax );
call stdout.puti16;
pop( eax );

// If the word value is in a 16-bit register
// then you can use code like the following:

push( eax ); // Assume wordVar is in AX
call stdout.puti16;

```



```
stdout.puti16Size( w:word; width:int32; fill:char );
```

This function writes the 16-bit signed integer value you pass to the standard output using the width and fill values as specified above.

HLA high-level calling sequence examples:

```
stdout.puti16Size( wordVar, width, padChar );
```

HLA low-level calling sequence examples:

```
// If "wordVar" and "padChar" are not one of the last three
// bytes on a page of memory, you can do this:

push( (type dword wordVar) );
push( width );
push( (type dword padChar) );
call stdout.putil6Size;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

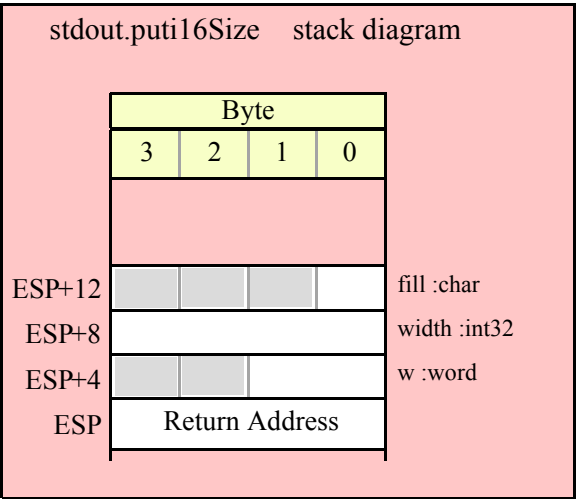
movzx( wordVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call stdout.putil6Size;

// If no registers are available, do something
// like the following code:

push( eax );
movzx( wordVar, eax );
push( eax );
push( width );
movzx( padChar, eax );
push( eax );
call stdout.putil6Size;
pop( eax );

// If "wordVar" is in a 16-bit register
// and "padChar" is in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:

push( eax ); // Assume wordVar is in AX
push( width );
push( ebx ); // Assume padChar is in BL
call stdout.putil6Size;
```



**stdout.puti32( d:dword );**

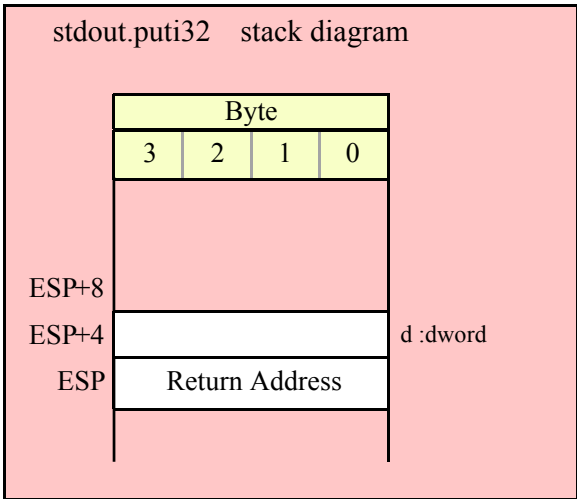
This function converts the 32-bit signed integer you pass as a parameter to a string and writes this string to the standard out using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
stdout.puti32( dwordVar );  
  
// If the dword is in a register (EAX):  
  
stdout.puti32( eax );
```

HLA low-level calling sequence examples:

```
push( dwordVar );  
call stdout.puti32;  
  
push( eax );  
call stdout.puti32;
```



```
stdout.puti32Size( d:dword; width:int32; fill:char );
```

This function writes the 32-bit value you pass as a signed integer to the standard output device using the width and fill values as specified above.

HLA high-level calling sequence examples:

```
stdout.putu32Size( dwordVar, width, ' ' );

// If the dword is in a register (EAX):

stdout.putu32Size( eax, width, cl );
```

HLA low-level calling sequence examples:

```
push( dwordVar );
push( width );
pushd( ' ' );
call stdout.putu32Size;

push( eax );
push( width );
push( ecx ); // fill char is in CL
call stdout.putu32Size;

// Assume fill char is in CH

push( eax );
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call stdout.putu32Size;

// Alternate method of the above

push( eax );
push( width );
sub( 4, esp );
mov( ch, [esp] );
call stdout.putu32Size;

// If the fill char is a variable and
// a register is available, try this code:

push( eax );
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call stdout.putu32Size;

// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( eax );
```



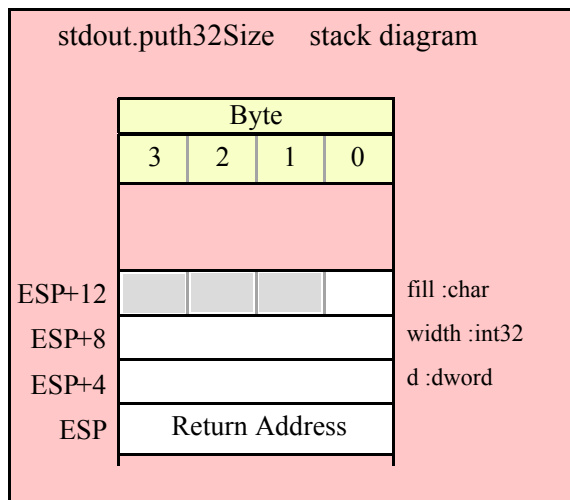
```

push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call stdout.putu32Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( eax );
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call stdout.puti32Size;

```



```
stdout.puti64( q:qword );
```

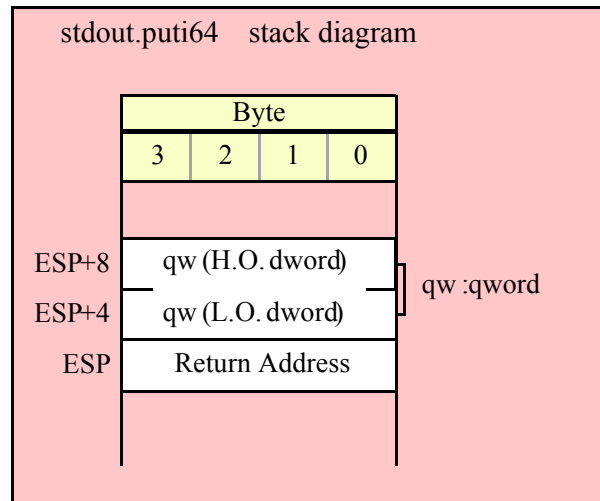
This function converts the 64-bit signed integer you pass as a parameter to a string and writes this string to the standard output using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
stdout.puti64( qwordVar );
```

HLA low-level calling sequence examples:

```
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));      // L.O. dword last
call stdout.puti64;
```



```
stdout.puti64Size( q:qword; width:int32; fill:char );
```

This function writes the 64-bit value you pass as a signed integer to the standard output using the width and fill values as specified above.

HLA high-level calling sequence examples:

```
stdout.puti64Size( qwordVar, width, ' ' );
```

HLA low-level calling sequence examples:

```
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
pushd( ' ' );
call stdout.puti64Size;
```

```
push( edx ); // Assume 64-bit value in edx:eax
push( eax );
push( width );
push( ecx ); // fill char is in CL
call stdout.puti64Size;
```

```
// Assume fill char is in CH
```

```
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call stdout.puti64Size;
```

```
// Alternate method of the above
```

```
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
```

```
push( width );
sub( 4, esp );
mov( ch, [esp] );
call stdout.puti64Size;

// If the fill char is a variable and
// a register is available, try this code:

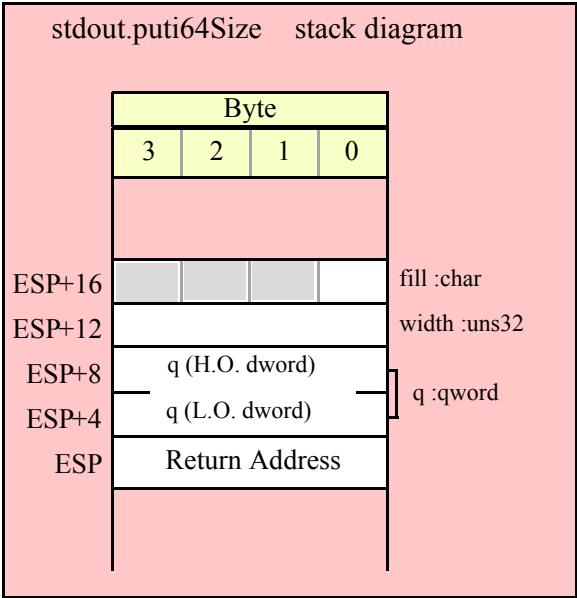
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));      // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call stdout.puti64Size;

// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));      // L.O. dword last
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call stdout.puti64Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));      // L.O. dword last
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call stdout.puti64Size;
```



```
stdout.puti128( l:lword );
```

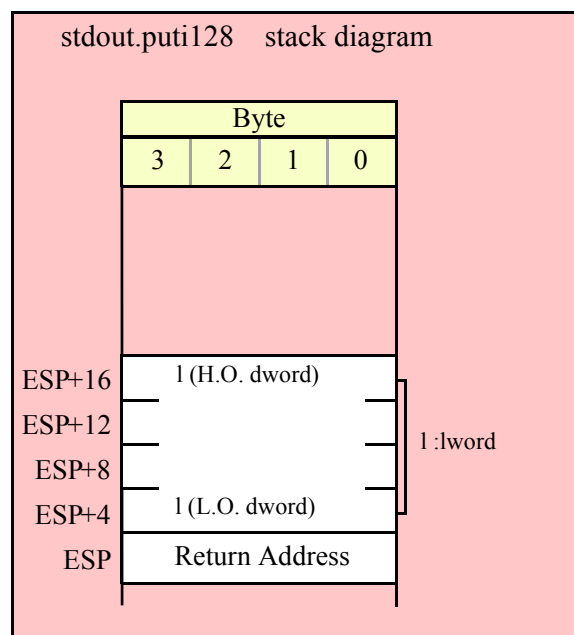
This function converts the 128-bit signed integer you pass as a parameter to a string and writes this string to the standard out using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
stdout.puti128( lwordVar );
```

HLA low-level calling sequence examples:

```
push( (type dword lwordVar[12]) ); // H.O. dword first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
call stdout.puti128;
```



```
stdout.puti128Size( l:lword; width:int32; fill:char );
```

This function writes the 128-bit value you pass as a signed integer to the standard output device using the width and fill values as specified above.

HLA high-level calling sequence examples:

```
stdout.puti128Size( lwordVar, width, ' ' );
```

HLA low-level calling sequence examples:

```
push( (type dword lwordVar[12]) ); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
```

```

push( width );
pushd( ' ' );
call stdout.puti128Size;

// Assume fill char is in CH

push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call stdout.puti128Size;

// Alternate method of the above

push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
push( width );
sub( 4, esp );
mov( ch, [esp] );
call stdout.puti128Size;

// If the fill char is a variable and
// a register is available, try this code:

push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call stdout.puti128Size;

// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
push( width );
push( (type dword fillChar) );      // Chance of page crossing!
call stdout.puti128Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

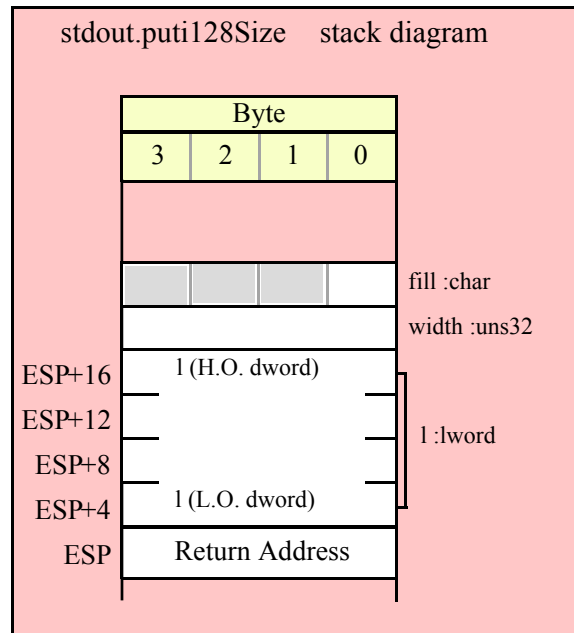
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
push( width );

```

```

sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call stdout.puti128Size;

```



## 31.9 Unsigned Integer Output Routines

These routines convert unsigned integer values to string format and write that string to the standard output device. The `stdout.putxxxSize` functions contain width and fill parameters that let you specify the minimum field width when outputting a value.

If the absolute value of width is greater than the number of print positions the value requires, then these functions output width characters to the standard out. If width is non-negative, then these functions right-justify the value in the output field; if value is negative, then these functions left-justify the value in the output field.

These functions print the fill character as the padding value for the extra print positions.

```
xxxSize( value, width, fill );
```

Assuming “value” requires five print positions, “width” is eight, and fill is “f” then the `xxxSize` functions produce the string

f	f	f	V	A	L	U	E
---	---	---	---	---	---	---	---

Assuming “value” requires five print positions, “width” is minus eight, and fill is “f” then the `xxxSize` functions produce the string

V	A	L	U	E	f	f	f
---	---	---	---	---	---	---	---

```
stdout.putu8 ( b:byte );
```

This function converts the eight-bit unsigned integer you pass as a parameter to a string and writes this string to the standard output device using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```

stdout.putu8( byteVar );

// If the character is in a register (AL):

stdout.putu8( al );

```

HLA low-level calling sequence examples:

```

// If "byteVar " is not one of the last three
// bytes on a page of memory, you can do this:

push( (type dword byteVar ) );
call stdout.putu8;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( byteVar , eax ); // Assume EAX is available
push( eax );
call stdout.putu8;

// If no register is available, do something
// like the following code:

push( eax );
movzx( byteVar , eax );
push( eax );
call stdout.putu8;
pop( eax );

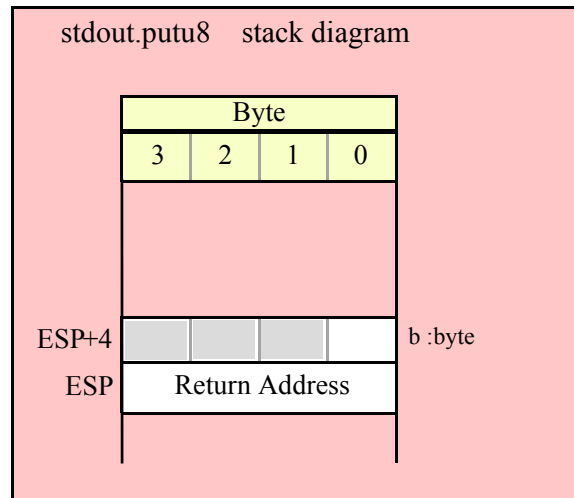
// If the character value is in al, bl, cl, or dl
// then you can use code like the following:

push( eax ); // Assume byteVar is in AL
call stdout.putu8;

// If the byte value is in ah, bh, ch, or dh
// you'll have to use code like the following:

xchg( al, ah ); // Assume byteVar is in AH
push( eax );    // It's now in AL
xchg( al, ah ); // Restore al/ah
call stdout.putu8;

```



```
stdout.putu8Size( b:byte; width:int32; fill:char );
```

This function writes the unsigned eight-bit value you pass to the standard output using the width and fill values as specified above.

HLA high-level calling sequence examples:

```
stdout.putu8Size( byteVar, width, padChar );
```

HLA low-level calling sequence examples:

```
// If "byteVar" and "padChar" are not one of the last three
// bytes on a page of memory, you can do this:
```

```
push( (type dword byteVar) );
push( width );
push( (type dword padChar) );
call stdout.putu8Size;
```

```
// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:
```

```
movzx( byteVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call stdout.putu8Size;
```

```
// If no registers are available, do something
// like the following code:
```

```
push( eax );
movzx( byteVar, eax );
push( eax );
push( width );
movzx( padChar, eax );
push( eax );
```



```

call stdout.putu8Size;
pop( eax );

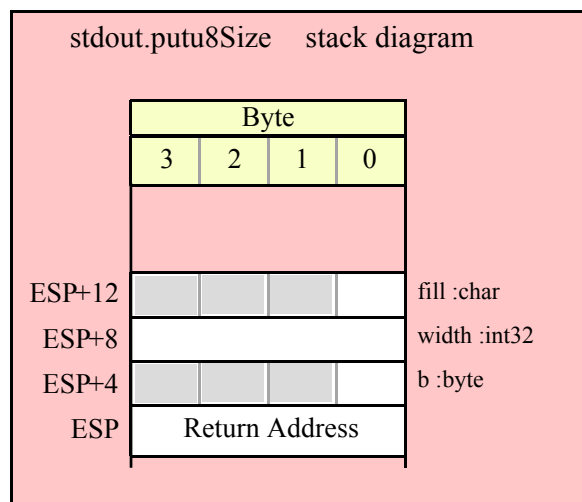
// If "byteVar" or "padChar" are in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:

push( eax );    // Assume byteVar is in AL
push( width );
push( ebx );    // Assume padChar is in BL
call stdout.putu8Size;

// Do the following if the characters are
// in AH, BH, CH, or DH:

xchg( al, ah );    // Assume byteVar is in AH
xchg( bl, bh );    // Assume padChar is in BH
push( eax );
push( width );
push( ebx );
xchg( al, ah );
xchg( bl, bh );
call stdout.putu8Size;

```



**stdout.putu16( w:word );**

This function converts the 16-bit unsigned integer you pass as a parameter to a string and writes this string to the standard output device using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```

stdout.putu16( wordVar );

// If the word is in a register (AX):

stdout.putu16( ax );

```

HLA low-level calling sequence examples:

```

// If "wordVar " is not one of the last three
// bytes on a page of memory, you can do this:

push( (type dword wordVar) );
call stdout.putul6;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

movzx( wordVar, eax ); // Assume EAX is available
push( eax );
call stdout.putul6;

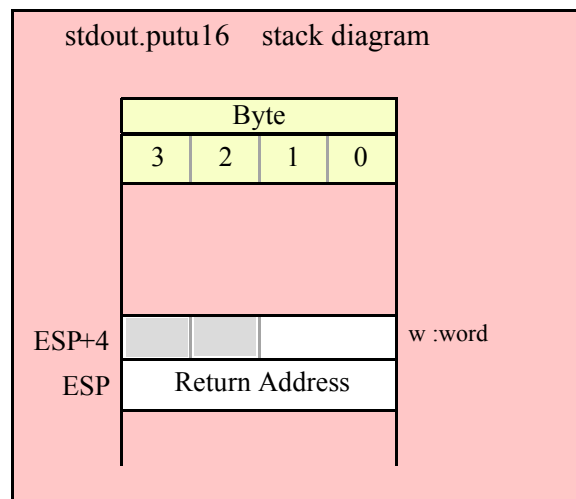
// If no register is available, do something
// like the following code:

push( eax );
movzx( wordVar, eax );
push( eax );
call stdout.putul6;
pop( eax );

// If the word value is in a 16-bit register
// then you can use code like the following:

push( eax ); // Assume wordVar is in AX
call stdout.putul6;

```



```
stdout.putul6Size( w:word; width:int32; fill:char );
```

This function writes the unsigned 16-bit value you pass to the standard out using the width and fill values as specified above.

HLA high-level calling sequence examples:

```
stdout.putul6Size( wordVar, width, padChar );
```

HLA low-level calling sequence examples:

```
// If "wordVar" and "padChar" are not one of the last three
// bytes on a page of memory, you can do this:

push( (type dword wordVar) );
push( width );
push( (type dword padChar) );
call stdout.putul6Size;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

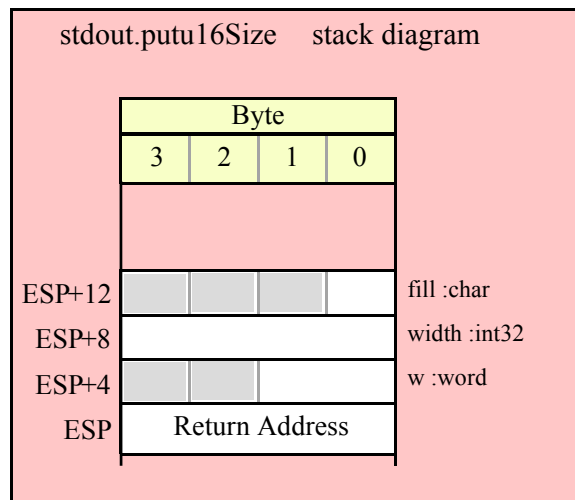
movzx( wordVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call stdout.putul6Size;

// If no registers are available, do something
// like the following code:

push( eax );
movzx( wordVar, eax );
push( eax );
push( width );
movzx( padChar, eax );
push( eax );
call stdout.putul6Size;
pop( eax );

// If "wordVar" is in a 16-bit register
// and "padChar" is in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:

push( eax ); // Assume wordVar is in AX
push( width );
push( ebx ); // Assume padChar is in BL
call stdout.putul6Size;
```



```
stdout.putu32( d:dword );
```

This function converts the 32-bit unsigned integer you pass as a parameter to a string and writes this string to the standard out using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
stdout.putu32( dwordVar );
```

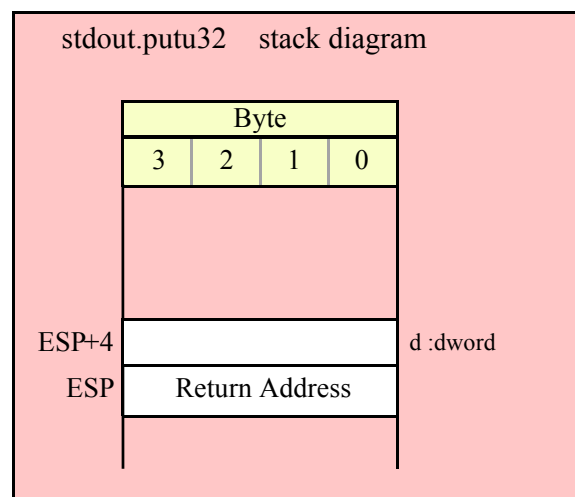
```
// If the dword is in a register (EAX):
```

```
stdout.putu32( eax );
```

HLA low-level calling sequence examples:

```
push( dwordVar );
call stdout.putu32;
```

```
push( eax );
call stdout.putu32;
```



```
stdout.putu32Size( d:dword; width:int32; fill:char );
```

This function writes the unsigned 32-bit value you pass to the standard out using the width and fill values as specified above.

HLA high-level calling sequence examples:

```
stdout.putu32Size( dwordVar, width, ' ' );

// If the dword is in a register (EAX):

stdout.putu32Size( eax, width, cl );
```

HLA low-level calling sequence examples:

```
push( dwordVar );
push( width );
pushd( ' ' );
call stdout.putu32Size;

push( eax );
push( width );
push( ecx ); // fill char is in CL
call stdout.putu32Size;

// Assume fill char is in CH

push( eax );
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call stdout.putu32Size;

// Alternate method of the above

push( eax );
push( width );
sub( 4, esp );
mov( ch, [esp] );
call stdout.putu32Size;

// If the fill char is a variable and
// a register is available, try this code:

push( eax );
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call stdout.putu32Size;

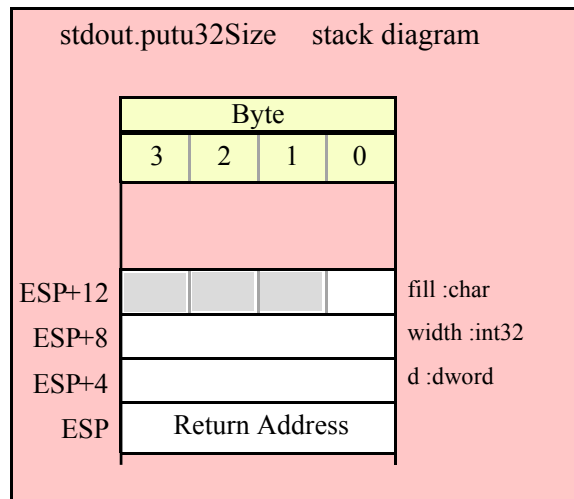
// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( eax );
```

```
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call stdout.putu32Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( eax );
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call stdout.putu32Size;
```



```
stdout.putu64( q:qword );
```

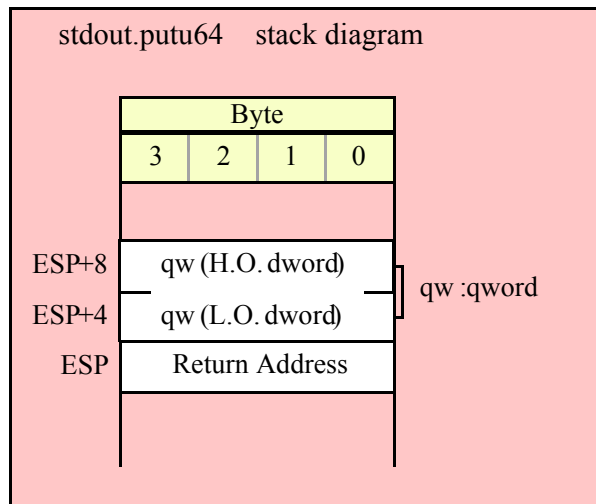
This function converts the 64-bit unsigned integer you pass as a parameter to a string and writes this string to the standard output device using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
stdout.putu64( qwordVar );
```

HLA low-level calling sequence examples:

```
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));      // L.O. dword last
call stdout.putu64;
```



```
stdout.putu64Size( q:qword; width:int32; fill:char );
```

This function writes the unsigned 64-bit value you pass to the output using the width and fill values as specified above.

HLA high-level calling sequence examples:

```
stdout.putu64Size( qwordVar, width, ' ' );
```

HLA low-level calling sequence examples:

```
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
pushd( ' ' );
call stdout.putu64Size;
```

```
push( edx ); // Assume 64-bit value in edx:eax
push( eax );
push( width );
push( ecx ); // fill char is in CL
call stdout.putu64Size;
```

```
// Assume fill char is in CH
```

```
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call stdout.putu64Size;
```

```
// Alternate method of the above
```

```
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
```

```

push( width );
sub( 4, esp );
mov( ch, [esp] );
call stdout.putu64Size;

// If the fill char is a variable and
// a register is available, try this code:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));      // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call stdout.putu64Size;

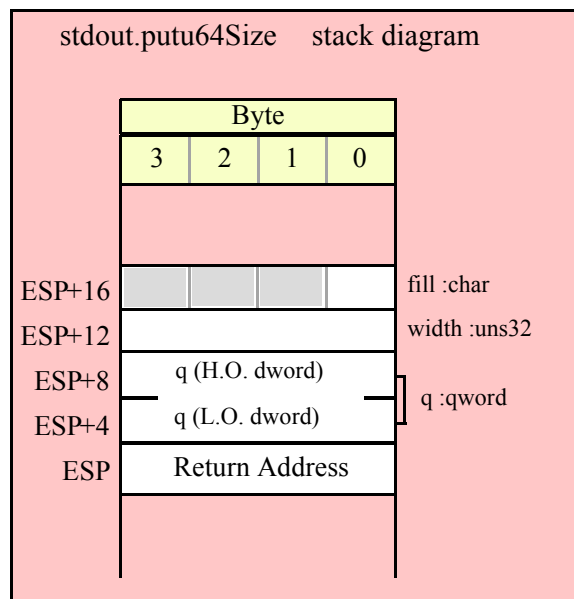
// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));      // L.O. dword last
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call stdout.putu64Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));      // L.O. dword last
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call stdout.putu64Size;

```





**stdout.putu128( l:ldword );**

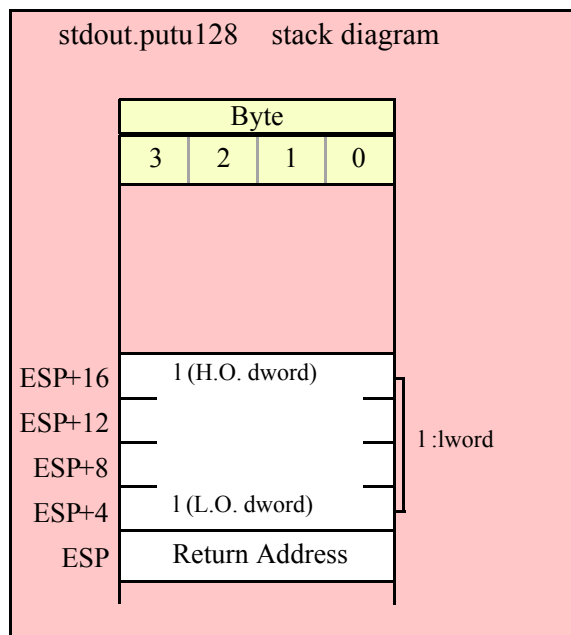
This function converts the 128-bit unsigned integer you pass as a parameter to a string and writes this string to the standard out using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
stdout.putu128( lwordVar );
```

HLA low-level calling sequence examples:

```
push( (type dword lwordVar[12]) ); // H.O. dword first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
call stdout.putu128;
```



**stdout.putu128Size( l:ldword; width:int32; fill:char )**

This function writes the unsigned 128-bit value you pass to the standard out using the width and fill values as specified above.

HLA high-level calling sequence examples:

```
stdout.putu128Size( lwordVar, width, ' ' );
```

HLA low-level calling sequence examples:

```
push( (type dword lwordVar[12]) ); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
```

```

push( width );
pushd( ' ' );
call stdout.putul28Size;

// Assume fill char is in CH

push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));    // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call stdout.putul28Size;

// Alternate method of the above

push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));    // L.O. dword last
push( width );
sub( 4, esp );
mov( ch, [esp] );
call stdout.putul28Size;

// If the fill char is a variable and
// a register is available, try this code:

push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));    // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call stdout.putul28Size;

// If the fill char is a variable and
// no register is available, here's one
// possibility:

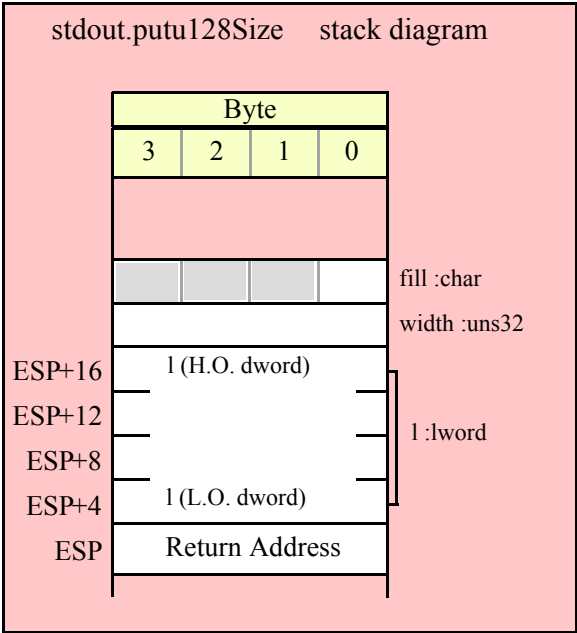
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));    // L.O. dword last
push( width );
push( (type dword fillChar) );    // Chance of page crossing!
call stdout.putul28Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));    // L.O. dword last
push( width );

```

```
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call stdout.putu128Size;
```



31.10 Floating Point Output Routines

The HLA standard output module provides several procedures you can use to write floating point values to the standard output device. The following subsections describe these routines.

31.10.1 Real Output Using Scientific Notation

The floating point numeric output routines translate the three different binary floating point formats to their string representation and then write this string to the standard output. There are two generic classes of these routines: those that convert their values to exponential/scientific notation and those that convert their string to a decimal form.

The `stdout.pute80`, `stdout.pute64`, and `stdout.pute32` routines convert their values to a string using scientific notation. These three routines each have two parameters: the value to output and the field width of the result. These routines produce a string with the following format:

s	i	.	f	f	f	f	f	E	±	x
---	---	---	---	---	---	---	---	---	---	---

s is a space for positive values, '-' for negative values  
i represents the integer portion of the mantissa  
ffff represents the fractional portion of the mantissa  
x is one or more base-10 exponent digits.

```
stdout.pute32( r:real32; width:uns32 );
```

This function writes the 32-bit single precision floating point value passed in *r* to the standard out using scientific/exponential notation. This procedure prints the value using width print positions in the output. width should have a minimum value of five for real numbers in the range 1e-9..1e+9 and a minimum value of six for all other values. Note that 32-bit extended precision floating point values support about 6-7 significant digits. So a width value that yeilds more than seven mantissa digits will produce garbage output in the low order digits of the number.

HLA high-level calling sequence examples:

```
stdout.pute32( r32Var, width );

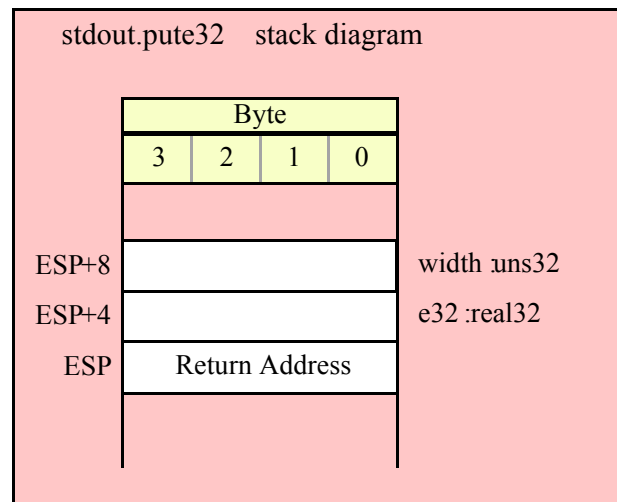
// If the real32 value is in an FPU register (ST0):

var
    r32Temp:real32;
    .
    .
    .
fstp( r32Temp );
stdout.pute32( r32Temp, 12 );
```

HLA low-level calling sequence examples:

```
push( (type dword r32Var) );
push( width );
call stdout.pute32;

sub( 4, esp );
fstp( (type real32 [esp]) );
pushd( 12 );
call stdout.pute32;
```



```
stdout.pute64( r:real64; width:uns32 );
```

This function writes the 64-bit double precision floating point value passed in *r* to the standard output using scientific/exponential notation. This procedure prints the value using width print positions in the output. width should have a minimum value of five for real numbers in the range 1e-9..1e+9 and a minimum value of six for all other values. Note that 64-bit double precision floating point values support about 15 significant digits. So a

width value that yeilds more than 15 mantissa digits will produce garbage output in the low order digits of the number.

HLA high-level calling sequence examples:

```
stdout.pute64( r64Var, width );

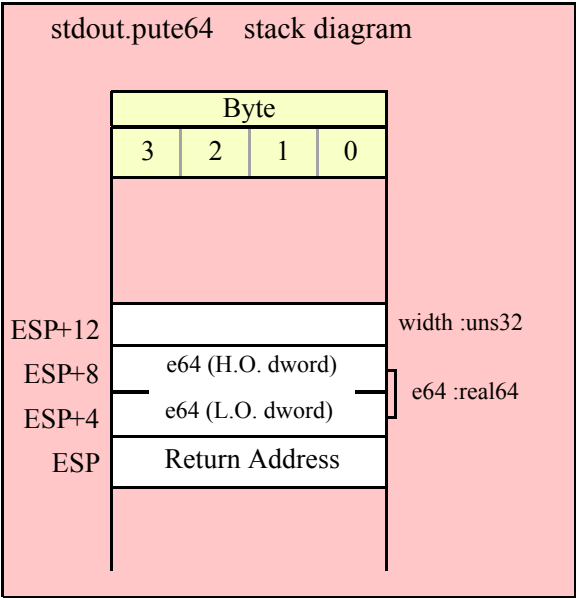
// If the real64 value is in an FPU register (ST0):

var
  r64Temp:real64;
  .
  .
  .
fstp( r64Temp );
stdout.pute64( r64Temp, 12 );
```

HLA low-level calling sequence examples:

```
push( (type dword r64Var[4]));
push( (type dword r64Var[0]));
push( width );
call stdout.pute64;

sub( 8, esp );
fstp( (type real64 [esp]) );
pushd( 12 );
call stdout.pute64;
```



```
stdout.pute80( r:real80; width:uns32 );
```

This function writes the 80-bit extended precision floating point value passed in r to the standard output device using scientific/exponential notation. This procedure prints the value using width print positions in the standard out. width should have a minimum value of five for real numbers in the range 1e-9..1e+9 and a minimum value of six for all other values. Note that 80-bit extended precision floating point values support

about 18 significant digits. So a width value that yeilds more than 18 mantissa digits will produce garbage output in the low order digits of the number.

HLA high-level calling sequence examples:

```
stdout.pute80( r80Var, width );

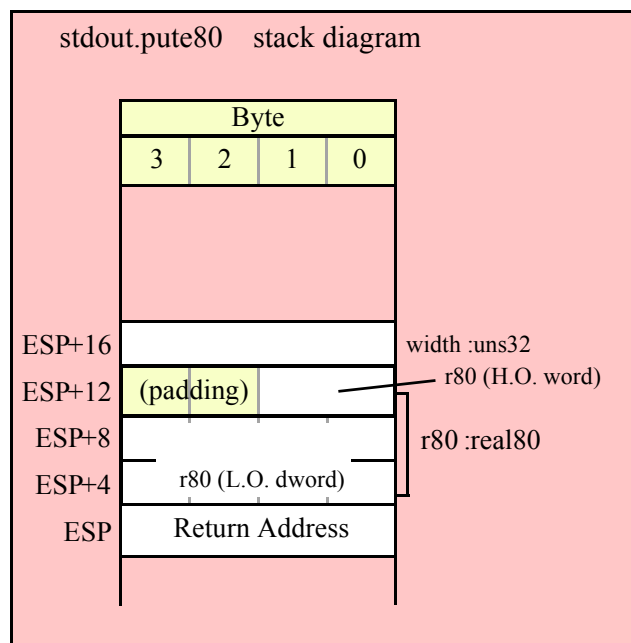
// If the real80 value is in an FPU register (ST0):

var
    r80Temp:real80;
.
.
.
fstp( r80Temp );
stdout.pute80( r80Temp, 12 );
```

HLA low-level calling sequence examples:

```
pushw( 0 ); // A word of padding.
push( (type word r80Var[8]) );
push( (type dword r80Var[4]) );
push( (type dword r80Var[0]) );
push( width );
call stdout.pute80;

sub( 12, esp );
fstp( (type real80 [esp]) );
pushd( 12 );
call stdout.pute80;
```



### 31.10.2 Real Output Using Decimal Notation

Although scientific (exponential) notation is the most general display format for real numbers, real numbers you display in this format are very difficult to read. Therefore, the HLA stdout module also provides a set of

functions that output real values using the decimal representation. Although you cannot (practically) use these decimal output routines for all real values, they are applicable to a wide variety of common numbers you will use in your programs.

These functions require four parameters: the real value to convert, the width of the converted value, the number of digit positions to the right of the decimal point, and a padding character. These functions write their values using the following string format:

s	i	i	i	.	f	f	f	f	f	f
---	---	---	---	---	---	---	---	---	---	---

s is a space for positive values, '-' for negative values  
i represents the integer portion of the mantissa  
ffffff represents the fractional portion of the mantissa

```
stdout.putr32( r:real32; width:uns32; decpts:uns32; pad:char );
```

This procedure writes a 32-bit single precision floating point value to the standard output as a string. The string consumes exactly width characters in the standard output. If the numeric output, using the specified number of positions to the right of the decimal point, is sufficiently small that the string representation would be less than width characters, then this procedure uses the value of pad as the padding character to fill the output with width characters. The number is right-justified within the output field (that is, this function prints the padding characters before the string representation of the number).

HLA high-level calling sequence examples:

```
stdout.putr32( r32Var, width, decpts, fill );
stdout.putr32( r32Var, 10, 2, '*' );

// If the real32 value is in an FPU register (ST0):

var
    r32Temp    :real32;
    .
    .
    .
fstp( r32Temp );
stdout.putr32( r32Temp, 12, 2, al );
```

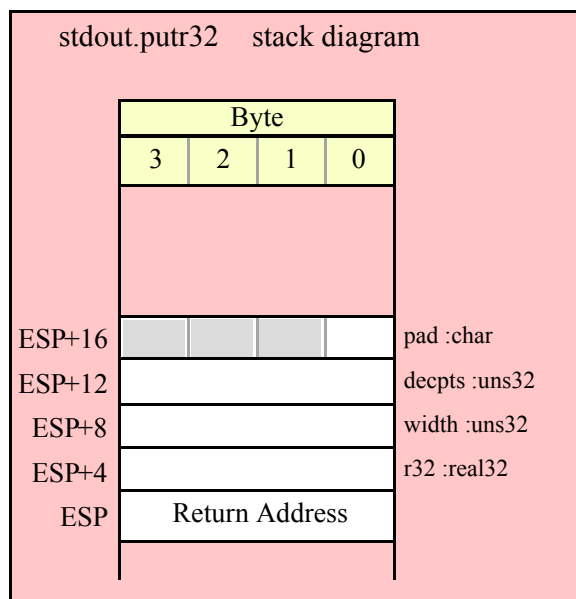
HLA low-level calling sequence examples:

```
push( (type dword r32Var) );
push( width );
push( decpts );
movzx( fill, eax );
push( eax );
call stdout.putr32;

push( (type dword r32Var) );
push( width );
push( decpts );
pushd( (type dword fill) ); // If no memory fault possible
call stdout.putr32;

sub( 4, esp );
fstp( (type real32 [esp]) );
pushd( 12 );
```

```
sub( 4, esp );
push( eax );
movzx( fill, eax ); // If memory fault were possible
mov( eax, [esp+4] ); // in above code.
pop( eax );
call stdout.putr32;
```



```
stdout.putr64(r:real64; width:uns32; decpts:uns32; pad:char);
```

This procedure writes a 64-bit double precision floating point value to the standard output device as a string. The string consumes exactly width characters in the output. If the numeric output, using the specified number of positions to the right of the decimal point, is sufficiently small that the string representation would be less than width characters, then this procedure uses the value of pad as the padding character to fill the output with width characters.

HLA high-level calling sequence examples:

```

stdout.putr64( r64Var, width, decpts, fill );
stdout.putr64( r64Var, 10, 2, '*' );

// If the real64 value is in an FPU register (ST0):

var
    r64Temp:real64;
.
.
.
fstp( r64Temp );
stdout.putr64( r64Temp, 12, 2, al );

```

HLA low-level calling sequence examples:

```
push( (type dword r64Var[4]) );
push( (type dword r64Var) );
push( width );
push( decpts );
```



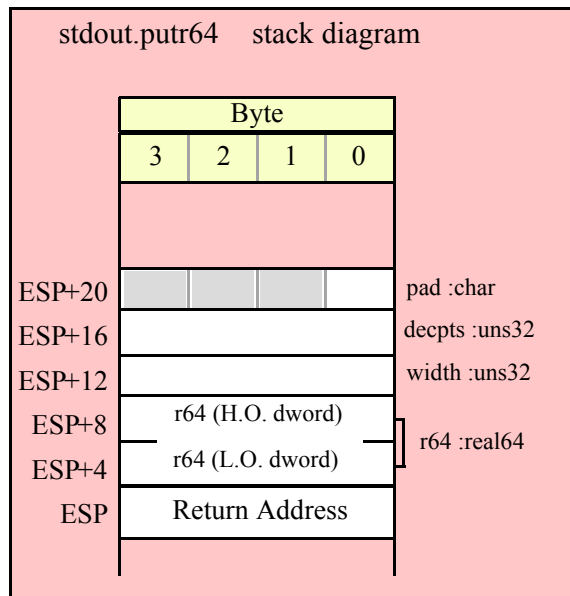
```

movzx( fill, eax );
push( eax );
call stdout.putr64;

push( (type dword r64Var[4]) );
push( (type dword r64Var) );
push( width );
push( decpts );
pushd( (type dword fill) ); // If no memory fault possible
call stdout.putr64;

sub( 8, esp );
fstp( (type real64 [esp]) );
pushd( 12 );
sub( 4, esp );
push( eax );
movzx( fill, eax );// If memory fault were possible
mov( eax, [esp+4] ); // in above code.
pop( eax );
call stdout.putr64;

```



**stdout.putr80( r:real80; width:uns32; decpts:uns32; pad:char);**

This procedure writes an 80-bit extended precision floating point value to the output as a string. The string consumes exactly width characters in the output. If the numeric output, using the specified number of positions to the right of the decimal point, is sufficiently small that the string representation would be less than width characters, then this procedure uses the value of pad as the padding character to fill the output with width characters.

HLA high-level calling sequence examples:

```

stdout.putr80( r80Var, width, decpts, fill );
stdout.putr80( r80Var, 10, 2, '*' );

// If the real80 value is in an FPU register (ST0):

var
    r80Temp:real80;

```

```

.
.
.
fstp( r80Temp );
stdout.putr80( r80Temp, 12, 2, al );

```

HLA low-level calling sequence examples:

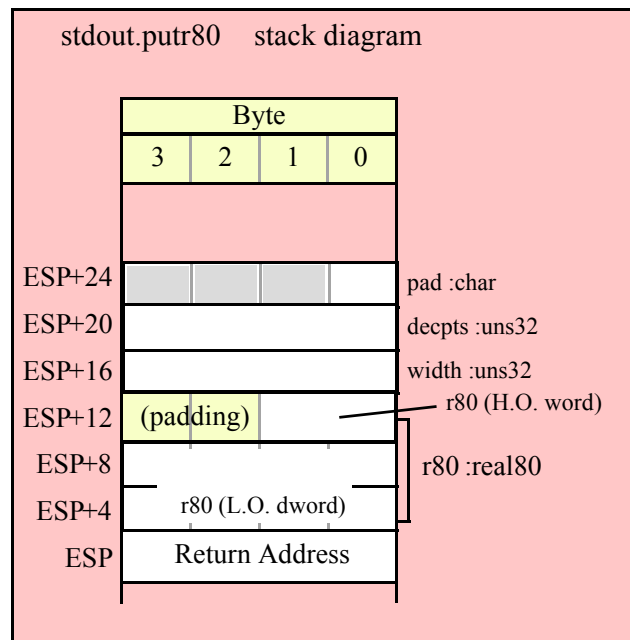
```

pushw( 0 );
push( (type word r80Var[8]) );
push( (type dword r80Var[4]) );
push( (type dword r80Var) );
push( width );
push( decpts );
movzx( fill, eax );
push( eax );
call stdout.putr80;

pushw( 0 );
push( (type word r80Var[8]) );
push( (type dword r80Var[4]) );
push( (type dword r80Var) );
push( width );
push( decpts );
pushd( (type dword fill) ); // If no memory fault possible
call stdout.putr80;

sub( 12, esp );
fstp( (type real80 [esp]) );
pushd( 12 );
sub( 4, esp );
push( eax );
movzx( fill, eax ); // If memory fault were possible
mov( eax, [esp+4] ); // in above code.
pop( eax );
call stdout.putr80;

```



## 31.11 Generic Standard Output Routine

```
stdout.put( list_of_items );
```

`stdout.put` is a macro that automatically invokes an appropriate `stdout` output routine based on the type of the parameter(s) you pass it. This is a very convenient output routine and is probably the `stdout` output call you will use most often in your programs. Keep in mind that this macro is not a single function call; instead, HLA translates this macro into a sequence of calls to procedures like `stdout.putu32`, `stdout.puts`, etc.

`stdout.put` is a macro that provides a flexible syntax for outputting data to the standard output device. This macro allows a variable number of parameters. For each parameter present in the list, `stdout.put` will call the appropriate routine to emit that data, according to the type of the parameter. Parameters may be constants, registers, or memory locations. You must separate each macro parameter with a comma.

Here is an example of a typical invocation of `stdout.put`:

```
stdout.put( "I=", i, " j=", j, nl );
```

The above is roughly equivalent to

```
stdout.puts( "I=" );
stdout.putu32( i );
stdout.puts( " j=" );
stdout.putu32( j );
stdout.newln();
```

This assumes, of course, that `i` and `j` are `int32` variables.

The `stdout.put` macro also lets you specify the minimum field width for each parameter you specify. To print a value using a minimum field width, follow the object you wish to print with a colon and the value of the minimum field width. The previous example, using field widths, could look like the following:

```
stdout.put( "I=", i:2, " j=", j:5, nl );
```

Although this example used the literal decimal constants two and five for the field widths, keep in mind that register values and memory value (integers, anyway) are perfectly legal here.

For floating point numbers you wish to display in decimal form, you can specify both the minimum field width and the number of digits to print to the right of the decimal point by using the following syntax:

```
stdout.put( "Real value is ", f:10:3, nl );
```

The `stdout.put` macro can handle all the basic primitive types, including boolean, unsigned (8, 16, 32, 64, 128), signed (8, 16, 32, 64, 128), character, character set, real (32, 64, 80), string, and hexadecimal (byte, word, dword, qword, lword).

There is a known "design flaw" in the `stdout.put` macro. You cannot use it to print HLA intermediate variables (i.e., non-local VAR objects). The problem is that HLA's syntax for non-local accesses takes the form "reg32:varname" and `stdout.put` cannot determine if you want to print reg32 using varname print positions versus simply printing the non-local varname object. If you want to display non-local variables you must copy the non-local object into a register, a static variable, or a local variable prior to using `stdout.put` to print it. Of course, there is no problem using the other `stdout.putXXXX` functions to display non-local VAR objects, so you can use those as well.



## 32 The HLA Standard Template Library

The following sections provide a basic description of some of the routines in the HLA Standard Template Library. Keep in mind that the HLA Standard Template Library is a work in progress and the following sections may not be totally up to date. The HLA Standard Template Library header file and source code is the final arbiter if there is a question how the routines operate.

Unless otherwise noted, you can assume that the Standard Library routines preserve all the general purpose registers. They generally do not preserve the flags.

### 32.1 Introduction to the HLA STL

The HLA Standard Template Library (STL) was designed to be similar to the C++ STL. The idea is not only to provide similar functionality to the C++ STL, but also to help make the transition from C++ to assembly language an easier process. Though the HLA STL is by no means an exact replicate of the C++ STL, the concepts are sufficiently close to allow someone to use the HLA STL in the same way they'd use the C++ STL without having to learn a new programming paradigm.

Though the HLA STL is especially easy to learn by those who are familiar with templates in C++, it's also a relatively straight-forward package to learn by those who are not C++ programmers. The HLA STL package provides convenient code for declaring dynamic arrays, queues, lists, lookup tables, and other advanced data structures. By using HLA STL code, you'll find it much easier to write advanced assembly language code taking advantage of these sophisticated data structures.

"Template" is a special C++ term that is effectively a synonym for *macro*<sup>1</sup>. Therefore, one big difference you'll find between the HLA STL and the HLA Standard Library is that there are not object files you link in with code that uses the HLA STL. The STL is simply a set of macros that you incorporate into your program by including the "stl.hhf" header file and then invoking the templates (macros) that interest you. Therefore, to use the HLA STL package, the first thing you must do is include the following statement in your HLA program:

```
#include( "stl.hhf" )
```

Note that HLA "stdlib.hhf" header file does not automatically include the STL header file. The STL and the HLA Standard Library are two separate packages and you must explicitly include "stl.hhf" to use the HLA STL facilities.

The HLA STL is a set of macros (templates) that create user-defined class objects when you invoke them. To a programmer, these macros look somewhat like user-defined types that you use in a type declaration section. For example, consider the following *vector* type declaration:

```
type
  int32Vector :stl.vector( int32 );
```

The principle difference between an STL type declaration and a standard type declaration is the fact that STL declarations are *parameterized*. STL types are abstract data types that usually *contain* some other type. A vector type, for example, is a dynamic array type, with each element of the vector being some base type (*int32* in the vector example above). It is possible to have vectors of 32-bit integers (*int32*), characters, strings, or any other built-in or user-defined data type. The parameter associated with an STL declaration specifies the underlying data type on which the new type is built. Consider the following two vector declarations:

```
type
  int32Vector :stl.vector( int32 );
  stringVector :stl.vector( string );
```

These two declarations create two new class types, an *int32* vector and a string vector, that one can use to declare integer and string vectors. It's important to realize that the vector template creates different types, not variables. It's also important to realize that vector types are different. That is, *int32Vector* and *stringVector*, although both vectors, are not compatible types.

---

1. Technically, this is not true, but we'll ignore the distinction in this document.

## 32.2 Type Declarations Created by a Template

Templates only create types, not variables. In order to create actual variable objects, you must declare such objects in an HLA *var*, *static*, or *storage* section (because all template types are classes, you cannot create initialized class objects in a readonly [or static] section).

Template expansions may only occur in an HLA *type* section at the global level of a program or unit. This is because the template expansion, in addition to creating the data type, emits the code for all the class methods and the VMT for the class. After a template expansion, the template will leave the program in the *type* declaration section, but keep in mind that internally, the template expands to code and data in addition to type declarations.

In addition to the user-specified type name, an STL declaration typically creates two or three other types during expansion. Most templates will also create the following types:

```
type
    p_name :pointer to name;
    name_cursor: pointer to XXXX;
```

where *name* is the user-specified type name (e.g., *int32Vector* and *stringVector* from the previous examples, yielding *int32Vector\_cursor* and *p\_int32Vector*). *XXXX* represents an unimportant type name for our purposes; Cursor types are opaque insofar as an HLA application will use a cursor type to pass data amongst template class methods without needing to know what the type actually references.

Some template types (e.g., *list* and *table*) also create a node type, declared as follows:

```
type
    name_node: record
        data:parameter_type;
        <<other_fields>>
    endrecord;
```

where *parameter\_type* is the type passed to the STL template as a parameter (e.g., *int32* and *string* in the current examples) and *other\_fields* represent some opaque fields that the class' methods reference and should be treated as private data by all other code.

Once you invoke a template in a type declaration section, you can create actual objects using the template's resulting type. For example, to create *int32Vector* and *stringVector* objects, you could use declarations like the following:

```
type
    myInt32v :int32Vector;
    myStringv :stringVector;
```

Note that you do not use an STL template when you define an actual variable. You use templates to create types and then you use those types you've created to declare variables.

## 32.3 Template Objects are Classes

Though it is not particularly apparent from the invocation of a template, you should realize that HLA STL templates create class types. When you declare a variable of some template type you've defined, you're creating a class object. Therefore, it helps if you're familiar with the HLA object-oriented programming paradigm (and, in particular, HLA classes, methods, class procedures, and class iterators).

Note that when you define a new type using an HLA template, that type definition also creates a set of methods, procedures, and iterators specific to that class. That is, a type declaration like the following:

```
type
    csetVector :stl.vector( cset );
```

does a lot more than simply define a type – it also expands to a lot of code to your source file. A typical template class may have 2-4 dozen methods, class procedures, and class iterators associated with the class. Each time you create a new class by expanding an STL template, you get a new copy of all those routines. Consider the following pair of type declarations:

```
type
```

```
int32Vector :stl.vector( int32 );
i32Vector :stl.vector( int32 );
```

Internally, these classes are exactly the same. Externally, however, they are different types. Therefore, the HLA STL will create a separate set of methods that are *absolutely identical* in everything but name for these two types. This is a waste of space. In general, you should only create one instance of a particular STL class, so that you only expand one set of methods, procedures, and iterators (and a virtual method table [VMT]) for that one class. If you really two different "vector of int32" types, you should consider a declaration like the following:

```
type
  i32v :stl.vector( int32 );
  int32Vector :i32v;
  i32Vector :i32v;
```

HLA only generates one set of methods/procedures/iterators and VMTs for the `i32v` class; the `int32Vector` and `i32Vector` classes share this code and VMT (which is reasonable, as the operations on `int32Vector` and `i32Vector` types will always be the same).

Note that the HLA STL macros emit *different* methods, procedures, and iterators for class objects with differing underlying types. For example, the `int32Vector` and `stringVector` types both need their own set of specific methods/procedures/iterators because those routines operate on completely different data types.

So keep in mind that everytime you expand an HLA STL template, you get a new set of routines associated with the corresponding class you've defined. Of course, you only have one set of routines for each class you create, regardless of how many instances (class variables) of that class you define. That is, declaring multiple variables does not cause the emission of multiple sets of methods; only defining *types* will do this.

## 32.4 Class Traits

A template trait is a compile-time or run-time value that provides some information about the type of the underlying class. The HLA STL defines several common trait objects that are testable within any STL type. Advanced programmers may use conditional assembly to test compile-time traits or actual machine instructions to test run-time traits. By utilizing traits, your code can behave differently, depending on the underlying (template) data type.

Though most programmers can use STL class types without ever worrying about traits, the availability of traits makes it possible to do some very sophisticated things with the HLA STL. Where traits might come in handy is when you're writing your own macros to which you pass different STL objects and you might need to generate different code (or emit an error message) depending on the traits the object supports. Also, you soon find out that it's possible to create an object with fewer traits than the default object supports. For example, if you're using some simple `int32 vector` types and you don't require any of the cursor capabilities, you can tell the HLA STL to construct an `int32 vector` type without cursor support (thus reducing the amount of code the template generates). However, if you pass one of these vector objects to a generic macro that works with vectors, the lack of cursor support could create a problem. Fortunately, traits solve this problem by letting that macro (or even some run-time code) test to see whether cursor support is present, generating an error (or otherwise handling the situation) if cursors are not available.

### 32.4.1 isSTL\_c Trait

The most fundamental trait associated with all template classes is the `isSTL_c` trait. This is a compile-time constant (`const` class declaration) that is defined and set to TRUE for all STL types. You can use the HLA compile-time `@defined` function to test whether or not the `isSTL_c` field is defined for a given class type. If this symbol is defined, then you can generally assume that the underlying class is an HLA STL class and you can test for any of the other STL traits<sup>2</sup>. Here's how you'd typically use this trait:

```
#if( @defined( someType.isSTL_c ) )
  << code to compile, knowing that this is an STL type >>
#endif
```

---

2. Of course, there is nothing stopping someone from defining this constant in some arbitrary non-STL class, but you can generally assume that someone won't hijack your program's logic by doing this.

## 32.4.2 Compile-Time Traits

If the *isSTL\_c* field is defined, then the class will also define three dword constants: *hierarchy\_c*, *capabilities\_c*, and *performance\_c*. These constants are bit maps with each (defined) bit position corresponding to some capability, or lack thereof, of the current class object. If the bit position contains one, then the class possesses the corresponding capability; if the bit position contains zero, then the class lacks that capability.

The hierarchical traits specify which subclasses are associated with a given type. The capability traits specify which general methods are available to a given class. The performance traits provide an indication of the performance of the methods available to a given class.

The STL defines the following constants (which are all values with a single bit set):

Hierarchical traits (testable in *hierarchy\_c*):

- `stl.isContainer_c`
- `stl.isRandomAccess_c`
- `stl.IsArray_c`
- `stl.isVector_c`
- `stl.isDeque_c`
- `stl.isList_c`
- `stl.isTable_c`

Capability traits (testable in *capabilities\_c*):

- `stl.supportsOutput_c`
- `stl.supportsCompare_c`
- `stl.supportsInsert_c`
- `stl.supportsRemove_c`
- `stl.supportsAppend_c`
- `stl.supportsPrepend_c`
- `stl.supportsSwap_c`
- `stl.supportsForEach_c`
- `stl.supportsrForEach_c`
- `stl.supportsCursor_c`
- `stl.supportsSearch_c`
- `stl.supportsElementSwap_c`
- `stl.supportsObjSwap_c`
- `stl.elementsAreObjects_c`

Performance traits (testable in *performance\_c*):

- `stl.fastInsert_c`
- `stl.fastRemove_c`
- `stl.fastAppend_c`
- `stl.fastPrepend_c`
- `stl.fastSwap_c`
- `stl.fastSearch_c`
- `stl.fastElementSwap_c`

As their category suggests, you use these constants to test particular bits in the *hierarchy\_c*, *capability\_c*, and *performance\_c* compile-time variables, respectively. For example, if you have an STL class and you want to determine if it is a *vector* class, you could use code like the following:

```
#if( (mySTLObject.hierarchy_c & stl.isVector_c) <> 0 )

    << you can assume mySTLObject is a vector object here >>

#endif
```

## 32.4.3 Run-Time Traits

If the *isSTL\_c* field is defined, then the class object provides three run-time dword variables containing various set bits that determine the characteristics of that class. These run-time traits are the same as the compile-time traits except, of course, you can test their values at run-time using machine instructions. These variables are



*hierarchy*, *capabilities*, and *performance*. They are run-time analogs to the compile-time constants mentioned in the previous section and are associated with the same set of trait constant, i.e.,

Hierarchical traits (testable in *hierarchy\_c*):

- `stl.isContainer_c`
- `stl.isRandomAccess_c`
- `stl.IsArray_c`
- `stl.isVector_c`
- `stl.isDeque_c`
- `stl.isList_c`
- `stl.isTable_c`

Capability traits (testable in *capabilities\_c*):

- `stl.supportsOutput_c`
- `stl.supportsCompare_c`
- `stl.supportsInsert_c`
- `stl.supportsRemove_c`
- `stl.supportsAppend_c`
- `stl.supportsPrepend_c`
- `stl.supportsSwap_c`
- `stl.supportsForEach_c`
- `stl.supportsrForEach_c`
- `stl.supportsCursor_c`
- `stl.supportsSearch_c`
- `stl.supportsElementSwap_c`
- `stl.supportsObjSwap_c`
- `stl.elementsAreObjects_c`

Performance traits (testable in *performance\_c*):

- `stl.fastInsert_c`
- `stl.fastRemove_c`
- `stl.fastAppend_c`
- `stl.fastPrepend_c`
- `stl.fastSwap_c`
- `stl.fastSearch_c`
- `stl.fastElementSwap_c`

Because these are run-time values, you must use 80x86 machine instructions to test for these trait values, e.g.,

```
test( stl.supportsCursor, mySTLObj.capabilities );
if( @nz ) then

    << execute code that uses the cursor methods in the object >>

endif;
```

## 32.4.4 Trait Constants

The following subsections define the meaning of each of the traits. Note that the term "true" means that the trait value is non-zero (and will have a single set bit, the bit position determined by the particular trait) while "false" means that the trait's value is zero.

### 32.4.4.1 `stl.isContainer_c` Trait

This bit is set in *hierarchy\_c* or *hierarchy* if the STL object is (known to be) a container object. Currently, almost all STL objects are containers so this trait will be true. (The only STL object that is not a container is the base object, and you'll generally not declare any STL objects using the base type).

A container is a type that holds elements of some other type. All common STL types are container types as they are all composite data types (e.g., arrays, lists, tables, and so on).

### 32.4.4.2 `stl.isRandomAccess_c` Trait

This bit is set in *heirarchy\_c* or *hierarchy* if the underlying STL object provides efficient (O(1) time) random access to the underlying type's elements. Vectors and dequeues are examples of objects that support random access as it takes the same amount of time to access any arbitrary element of these types.

### 32.4.4.3 `stl.isArray_c` Trait

This bit is set in *heirarchy\_c* or *hierarchy* if the underlying class is an array object. Currently, this value is true for *vector* and *deque* types.

### 32.4.4.4 `stl.isVector_c` Trait

This bit is set in *heirarchy\_c* or *hierarchy* if the underlying class is a *vector* class type.

### 32.4.4.5 `stl.isDeque_c` Trait

This bit is set in *heirarchy\_c* or *hierarchy* if the underlying class is a *deque* class type.

### 32.4.4.6 `stl.isList_c` Trait

This bit is set in *heirarchy\_c* or *hierarchy* if the underlying class is a *list* class type.

### 32.4.4.7 `stl.isTable_c` Trait

This bit is set in *heirarchy\_c* or *hierarchy* if the underlying class is a *table* class type.

### 32.4.4.8 `stl.supportsOutput_c` Trait

This bit is set in *capabilities\_c* or *capabilities* if the underlying class supports a *toString* method that the HLA Standard Library can employ in macro invocations such as *stdout.put* or *fileio.put* to write the class' data to some output stream. By default, this value is false. If you provide a *toString* method for a given data type you define, then you'll set this constant to true.

### 32.4.4.9 `stl.supportsCompare_c` Trait

This bit is set in *capabilities\_c* or *capabilities* if the underlying class supports the *isEqual*, *isLess*, and *isLessEqual* methods. Some classes may only support an *isEqual* method, in which case the *supportsCompare\_c* trait will be false; you may test for *isEqual* by using the *@defined* compile-time function.

### 32.4.4.10 `stl.supportsInsert_c` Trait

This bit is set in *capabilities\_c* or *capabilities* if the class supports data insertion into an object. Generally, this implies that you have at least an *insertVal* and an *insertRef* method available. Other insertion methods may also be available, use *@defined* to test for their presence if you need to determine whether they exist for a particular class object. Not all class types accept the same parameter lists for their insert methods, thus limiting the generic usefulness of these methods (e.g., *table* insertions are based on strings rather than an integer index). You can test the *is\*\*\*\*\_c* traits (e.g., *isTable\_c*) to handle such cases.

### 32.4.4.11 `stl.supportsRemove_c` Trait

This bit is set in *capabilities\_c* or *capabilities* if it is possible to remove objects from an STL object at run-time. Some STL data types (e.g., tables) do not allow the removal of an object once it's been inserted into the object; such types will (obviously) set *supportsRemove\_c* to false.

### 32.4.4.12 `stl.supportsAppend_c` Trait

This bit is set in *capabilities\_c* or *capabilities* if it is possible to append a data element to the end of some STL object in memory. Some STL data types (e.g., tables) do not support the notion of a data sequence and, therefore, do not define an append operation. You can test this compile-time constant to check whether appending is a valid object before attempting it.

### 32.4.4.13 `stl.supportsPrepend_c` Trait

This bit is set in *capabilities\_c* or *capabilities* if it is possible to insert a data element at the front of some STL object in memory. Some STL data types (e.g., tables) do not support the notion of a data sequence and,

therefore, do not define a prepend operation. You can test this compile-time constant to check whether appending is a valid object before attempting it.

#### 32.4.4.14 **stl.supportsForEach\_c** and **supportsrForeach\_c** Traits

These two compile-time constants (testable in *capabilities\_c* and *capabilities*) tell you whether the template type supports a forward iterator (`ForEachElement`) or a reverse iterator (`rForEach`). Most STL types, by default, provide both types of iterators, even when the underlying data type is not a sequence (e.g., tables). For those data types that do not enforce an underlying sequence, the iterators will sequence through each of the object's elements, but the order of the sequence is undefined.

#### 32.4.4.15 **stl.supportsCursor\_c** Trait

Cursors are special opaque pointer objects that STL methods use to provide access to the underlying objects of some STL type. If an STL type supports cursors and operations on the type via those cursors, then this trait will be true for that type. This bit is set in *capabilities\_c* or *capabilities* if the class supports cursors.

#### 32.4.4.16 **stl.supportsSearch\_c** Trait

(to be defined; unused as this is being written.)

#### 32.4.4.17 **stl.supportsElementSwap\_c** Trait

This bit is set in *capabilities\_c* or *capabilities* if there is a *swapElements* method for the underlying class type. This method physically swaps the data between two elements of the STL type. This operation is not permitted for certain data types (e.g., tables), in which case the method will not exist and this trait will contain false.

#### 32.4.4.18 **stl.supportsObjSwap\_c** Trait

This bit is set in *capabilities\_c* or *capabilities* if the *swapObj* method is present. *swapObj* will completely swap the values of two STL variables (whole objects, not elements of those objects).

#### 32.4.4.19 **stl.elementsAreObjects\_c** Trait

This bit is set in *capabilities\_c* or *capabilities* if the elements of a given STL type are themselves class objects. This trait is set to false if the underlying data type is something other than a class. You may test this constant, for example, to determine if you should call constructors or destructors for each object created for an STL container class.

#### 32.4.4.20 **stl.fastInsert\_c** Trait

This bit is set in *performance\_c* or *performance* if the class supports insertion and insertion can be (typically) done in  $O(1)$  time. It will be set to false if the class does not support insertion or executes slowly (generally  $O(\lg n)$ ,  $O(n)$ , or worse, time is considered "not fast").

#### 32.4.4.21 **stl.fastRemove\_c** Trait

This bit is set in *performance\_c* or *performance* if the class supports element removal and removal can be (typically) done in  $O(1)$  time. It will be set to false if the class does not support removal or executes slowly (generally  $O(\lg n)$ ,  $O(n)$ , or worse, time is considered "not fast").

#### 32.4.4.22 **stl.fastAppend\_c** Trait

This bit is set in *performance\_c* or *performance* if the class supports element append and append can be (typically) done in  $O(1)$  time. It will be set to false if the class does not support append or executes slowly (generally  $O(\lg n)$ ,  $O(n)$ , or worse, time is considered "not fast").

#### 32.4.4.23 **stl.fastPrepend\_c** Trait

This bit is set in *performance\_c* or *performance* if the class supports element prepend and prepend can be (typically) done in  $O(1)$  time. It will be set to false if the class does not support prepend or it executes slowly (generally  $O(\lg n)$ ,  $O(n)$ , or worse, time is considered "not fast").

### 32.4.4.24 **stl.fastSwap\_c Trait**

This bit is set in *performance\_c* or *performance* if the class supports whole object swap and swapping can be (typically) done in  $O(1)$  time. It will be set to false if the class does not support object swap or it executes slowly (generally  $O(\lg n)$ ,  $O(n)$ , or worse, time is considered "not fast").

### 32.4.4.25 **stl.fastSearch\_c Trait**

(as this was being written, this trait was undefined.)

### 32.4.4.26 **stl.fastElementSwap\_c Trait**

This bit is set in *performance\_c* or *performance* if the class supports element swap and swapping can be (typically) done in  $O(1)$  time. It will be set to false if the class does not support element swap or it executes slowly (generally  $O(\lg n)$ ,  $O(n)$ , or worse, time is considered "not fast").

## 32.4.5 Other Run-Time Traits

All objects possess two run-time fields: *typeName* and *isAlloc*. The *typeName* field is a string that provides the actual name of the object. For example, given a declaration like the following:

```
type
  int32Vector :stl.vector( int32 );
```

then the *typeName* field will be initialized with the string "int32Vector". The *typeName* variable should be treated as a read-only object; you should not modify the pointer or the string data associated with it (actually, the string data is in a read-only section, so you cannot modify it, but you should modify the string pointer, either).

The *isAlloc* field will contain true if the object has been allocated on the heap, it will contain false if this object was not allocated on the heap. The destructor method (destroy) uses this field to determine whether it needs to deallocate storage when the object is deleted. You may read the value of this field, but you must not change it.

Container objects (which is all STL objects at this point) have two additional fields: *numElements* and *containerName*. The *numElements* field is a un32 variable that specifies the number of objects contained within the container (e.g., the number of *vector* elements or *list* nodes). You must not modify this field; treat it as a read-only object; indeed, there is a *getSize* method that you can use to retrieve the value of this field. You should use the *getSize* method and avoid accessing this field directly.

The *containerName* field is a string that specifies the container type. This will typically be a string like "vector", "deque", "list", or "table". You should treat this as a read-only field.

The *arrayContainer*, *vector*, *deque*, *list*, and *table* classes all contain their own private data fields. You should treat all these fields as opaque – that is, private to the class – and you should not modify or even read their values. Where necessary, these classes will provide accessor functions that return the values of these data fields.

## 32.5 The Vector Template

(to be written)

## 32.6 The Deque Template

(to be written)

## 32.7 The List Template

(to be written)

## 32.8 The Table Template

## 33 The Strings Module (strings.hhf)

HLA provides a sophisticated string handling package. The string data type has been carefully designed for high performance operations and there are lots of routines that perform almost every imaginable standard operation on the string data.

Note: be sure to read the chapter on "Passing Parameters to Standard Library Routines" (parmpassing.rtf) before reading this chapter.

**A Note About the FPU:** The Standard Library code makes occasional use of the FPU, particularly when converting between real and string formats and when computing certain mathematical functions. You should exercise caution when using MMX instructions in a program that makes use of the Standard Library. In particular, you should ensure that you are always in FPU mode (by executing an EMMS instruction) after you are finished using MMX instructions. Better yet, you should avoid the MMX instruction set altogether and use the improved SSE instruction set that accomplish the same tasks (and doesn't disturb the FPU).

### 33.1 The HLA String Data Type

The first place to start is with the discussion of the string data type itself. A string variable is nothing more than a four-byte pointer that points at the actual string data. So anytime you pass a string by value to a procedure or method, you're actually passing a pointer value. Note that taking the address of a string variable (with the LEA instruction) takes the address of the pointer, not the address of the actual character data. Therefore, if you are calling a routine that expects the address of some character data in a register, you would normally *move* the contents of a string variable into that register, not load the address of that string variable. For example, the *atoi* routine (see the chapter on conversions) expects a pointer to a string variable in the ESI register. If you wish to pass the address of the first character of a string in ESI, you would use the "mov(s,esi);" instruction, not "lea(esi,s);".

The HLA Standard Library makes a couple of important assumptions about where string variables are pointing. First, and most important, string variables must always point at a buffer that is an even multiple of four bytes long. Many string operations move double words, rather than bytes, around to improve performance. If the buffer is not an even multiple of four bytes long, some data transfers may inadvertently wipe out data adjacent to the string buffer or, worse still, cause a general protection fault.

The second assumption the HLA Standard Library makes is that the string data is prefaced by two dword objects. The first (at offset -8 from the beginning of the character data) contains the maximum number of characters that can be stored into this string (not counting a zero terminating byte). This value is fixed when storage is allocated for the string. The HLA string routines use this value to detect a string overflow condition.

The second dword object before the character data (at offset -4) is the current dynamic length of the string (that is, the actual number of characters currently in the string). Since the maximum and dynamic length fields are four bytes long, HLA supports (in theory) strings whose lengths are up to four gigabytes in length. In practice, of course, strings generally don't grow very large.

HLA strings always contain a zero terminating byte. Strictly speaking, this zero terminating byte is not absolutely necessary because the HLA string type includes a dynamic length field. As such, most of the HLA Standard Library routines (but not all) tend to ignore the zero terminating byte other than for use as a delimiter in the conversion routines. However, having this zero terminating byte allows you to pass HLA strings as parameters to Windows, FreeBSD, and Linux API functions and other functions that expect C/C++ style zero terminated strings.

Although not necessary for correct operation, HLA always aligns strings on a double word boundary. This allows certain string operations to run nearly twice as fast as they would if they were not aligned on a double word boundary.

To simplify access to the fields of a string, the string.hhf header file contain a record template you may use to access those fields in a structured fashion. This structure has the following definition:

```
type
    strRec: record := -8;

        maxlen:    int32;
        length:    int32;
        strData:    char[12];

    endrecord;
```

(The index value after the `char` type is arbitrary.)

For example, suppose you have a string variable `s` and you wish to know the current length of this string. You could obtain the length as follows:

```
mov( s, esi );
mov( (type str.strRec [esi]).length, eax );
```

(Note that the `str.strRec` type definition appears within the `str` namespace, hence the "str." prefix).

As a general rule, you should always use `str.alloc` (or some routine that winds up calling `str.alloc`) to allocate storage for string variables. If you must allocate the storage yourself, be sure the storage allocation follows all the rules specified earlier.

Consider what HLA does when you declare an initialized string object as follows:

```
static
  s :string := "SomeString";
```

One might be tempted to think that HLA allocates the string data as part of the `s` variable. In fact, this is not the case. HLA places the actual string data (including the length values, terminating byte, and any necessary padding bytes) *somewhere else* and then initializes the `s` object with the address of data data (appearing elsewhere). There is no direct way by only referencing `s` at compile-time, to treat the address of this string object as a constant. This feature would be useful, for example, for initializing string fields of a record constant with the address of the actual string data.

The HLA Standard Library `strings.hhf` header file provides a macro that lets you declare string constants and attach a label to the first character of that string (which is the address you generally want to assign to a string variable or field). You use this macro almost like the string data type, except you also supply a literal string constant argument, e.g.,

```
static
  s :str.constant( "SomeString" );
```

This creates a string object in memory with `s`'s address corresponding to the first character of the string object. Note that `s` is not an HLA string; remember, an HLA string is a pointer to a string object. The address of `s` is what would normally appear in a string variable. Now consider the following code:

```
type
  r:record
    s:string;
    b:byte;
  endrecord;

static
  somestr :str.constant( "SomeString" );
  a :r := r:[ &somestr, 0 ];
```

This initializes the `s` field of `a` with a pointer to the string containing the characters "SomeString". This is the proper way to initialize a string field of a record. *Note that HLA will accept the following without complaint, but it is not correct:*

```
static
  somestr :string := "SomeString";
  a :r := r:[ &somestr, 0 ];
```

This example initializes the `s` field of `a` with the address of `somestr`. But this is not string data, rather, it's the address of some string data. Therefore, this code initializes field `s` with a pointer to the pointer of some character data, rather than the pointer to the character data (which is what you probably want).

Here's the implementation of the `str.constant` macro, just in case you're wondering how it works:

```
// str.constant( literal_constant )
```

```
//
// This macro creates a string constant object whose address
// you may assign to a string variable. This is useful, for
// example, when initializing fields of a record with string data.

#macro constant( __strconst ):__strname,__padding;
    forward( __strname );
    align(4);
    dword @length( __strconst ), @length( __strconst );
    __strname:char; @nostorage;
    byte __strconst, 0;
    ?__padding := ((4-(@length(__strconst)+1)mod 4))mod 4);
    #while( __padding > 0 )

        byte 0;
        ?__padding -= 1;

    #endwhile

#endmacro;
```

## 33.2 String Allocation Macros and Functions

The functions and macros in this group deal with allocating storage for HLA strings.

**#macro str.strvar( size )**

*str.strvar* is a macro that will statically allocate storage for a string in the STATIC variable declaration section (you cannot use *str.strvar* in any of the other variable declaration sections, including the other static sections: READONLY, and STORAGE; you can *only* use it in the STATIC section). This macro emits the appropriate code to initialize a string pointer variable with the address of appropriate string storage that has sufficient room to hold a string of at least *size* characters (*size* is the parameter passed to this macro).

Example:

```
static
    StaticString: str.strvar( 32 );
```

Since the storage is statically allocated for *StaticString*, there is no need to call *str.alloc* or any other string/memory allocation procedure to allocate storage for this variable.

**procedure str.init( var b:var; numBytes:dword ); @returns( "eax" );**

This function initializes a block of memory for use as a string object. It takes the address of the buffer variable *b* and aligns this address on a dword boundary. Then it initializes the *maxlen*, *length*, and zero terminating byte fields at the resulting address. Finally, it returns a pointer to the newly created string object in EAX. The *numBytes* field specifies the size of the entire buffer area, not the desired maximum length of the string. The *numBytes* field must be 16 or greater, else this routine will raise an *ex.ValueOutOfRange* exception. Note that string initialization may consume as many as 15 bytes (up to three bytes to align the address on a dword boundary, four bytes for the *maxlen* field, four bytes for the *length* field, and the string data area must be a multiple of four bytes long (including the zero terminating byte). This is why the *numBytes* field must be 16 or greater. Note that this function initializes the resulting string to the empty string. The *maxlen* field will contain the maximum number of characters that you can store into the resulting string after subtracting the zero terminating byte, the sizes of the length fields, and any alignment bytes that were necessary.

HLA high-level calling sequence examples:

```
var
    strPtr:string;
    buffer:char [128 ];
```

```

    .
    .
    .
    str.init( buffer, 128 );
    mov( eax, strPtr );

```

HLA low-level calling sequence examples:

```

lea( eax, buffer );// Must push address of buffer object.
push( eax );
pushd( 128 );
call str.init;

```

### 33.3 String Length Calculations

As noted earlier, HLA format strings keep the current dynamic length in the four bytes immediately before the first byte of character data in the string object. In general, it's bad programming practice to assume anything about the internal data structure of a data type such as a string. However, the location of the HLA string field is well-known and just about everybody (even those who know better) directly access the string length field of the string data structure, so there is no way that this can ever change at this point. Therefore, you can feel fairly safe computing the length of a string using the length field of the *str.strRec* record data type. The typical way this is done is to load the string pointer value into a register and obtain the length as follows:

```

mov( someStr, ebx );
mov( (type str.strRec [ebx]).length, eax );

```

It remains a bad idea to access the length field at the fixed numeric offset -4 from the start of the string. Always use the *str.strRec* data type if you want to access the string length field.

For those who want a bonafide function that does the job, the HLA standard library does provide a string function (*str.length*) that you can call to fetch the length of a string object. The advantage of an actual string function is that you can take its address and do other things with it that can only be done with an actual procedure.

```
procedure str.length( src:string ); @returns( "eax" );
```

This function returns the current dynamic length of the string you pass as an argument.

HLA high-level calling sequence examples:

```

str.length( someStr );
mov( eax, lengthOfString );

```

HLA low-level calling sequence examples:

```

push( someStr );
call str.length;
mov( eax, lengthOfString );

```

### 33.4 String Assignment

The HLA standard library contains several functions you can use to copy string data from one location to another. These functions copy data from HLA strings to other HLA strings, from zero-terminated strings to HLA strings, and from other data objects to HLA strings. Some of them copy data to existing (preallocated) string objects and some of them allocate new storage for strings on the heap.



```
procedure str.a_cpy( src:string ); @returns( "(type string eax)" );
```

This function copies the string data from *src* to a new string object it allocates on the heap and returns a pointer to the new string object in EAX. This function will raise an exception if *src* is NULL, *src* is an invalid pointer, or it cannot allocate sufficient storage.

HLA high-level calling sequence examples:

```
str.a_cpy( someStr );
mov( eax, newStr );
```

HLA low-level calling sequence examples:

```
push( someStr );
call str.a_cpy;
mov( eax, newString );
```

```
procedure str.cpy( src:string; dest:string );
```

This function copies the string data from *src* to *dest*. The *dest* argument must point at an allocated string object in writeable storage that is large enough to hold a copy of the *src* string data. This function will raise an exception if *src* or *dest* is NULL or is an invalid pointer. It will also raise an exception if the string object pointed at by *dest* is too small to hold a copy of *src*'s data.

HLA high-level calling sequence examples:

```
str.cpy( someStr, destStr );
```

HLA low-level calling sequence examples:

```
push( someStr );
push( destStr );
call str.cpy;
```

```
procedure str.a_cpyz( zstr:zstring ); @returns( "(type string eax)" );
```

This function copies (and converts) the zero-terminated string data from *zstr* to a new HLA string object it allocates on the heap and returns a pointer to the new string object in EAX. This function will raise an exception if *zstr* is NULL, *zstr* is an invalid pointer, or it cannot allocate sufficient storage.

HLA high-level calling sequence examples:

```
str.a_cpyz( someZStr );
mov( eax, newHLAString );
```

HLA low-level calling sequence examples:

```
push( someZStr );
call str.a_cpy;
mov( eax, newHLAString );
```

```
procedure str.cpyz( zstr:zstring; dest:string );
```

This function copies (and converts) the zero-terminated string data from *zstr* to the HLA string *dest*. The *dest* argument must point at an allocated string object in writeable storage that is large enough to hold a copy of the *zstr* string data. This function will raise an exception if *zstr* or *dest* is NULL or is an invalid pointer. It will also raise an exception if the string object pointed at by *dest* is too small to hold a copy of *zstr*'s data.

HLA high-level calling sequence examples:

```
str.cpyz( someZStr, destHLStr );
```

HLA low-level calling sequence examples:

```
push( someZStr );
push( destHLAstr );
call str.cpyZ;
```

## 33.5 Substring Functions

The HLA Standard Library provides four families of functions that extract a portion of some string (that is, a substring): the *substr*, *first*, *last*, and *truncate* families of functions. These functions differ in how they compute the starting index of the substring to extract and, in the case of the *truncate* functions versus the other functions, how they determine the length of the substring to extract.

All of the substring extraction functions return a true/false status in the carry flag. These functions return with the carry flag set if the extracted substring is the length the caller specified (or, in the case of the *truncate* functions, the function truncated the specified number of characters). These functions return with the carry flag clear if the resulting substring is shorter than the length specified (or the number of characters truncated is fewer than specified) because the source string was too short. All of these functions return "@c" as their 'returns' value, so you can use these functions in a HLL-like control structure's boolean expression (e.g., in an 'if' statement) to test for success or failure.

```
procedure str.a_substr( src:string; index:dword; len:dword );  
@returns( "@c" );
```

This function extracts a substring of length *len* starting at character position *index* with the source string *src*. This function allocates storage for the substring data on the heap and returns a pointer to the new substring in the EAX register (note, however, that the function 'returns' value is "@c" and not "EAX"). It is the caller's responsibility to free the storage when it is done using the string data.

If the sum of *index*+*len* is greater than the length of the source string, this function returns with the carry flag cleared (equal to zero) to indicate that the substring was truncated. If *index*+*len* is less than or equal to the length of the source string, then this function returns with the carry flag set to one. Note that this function does not raise an exception if the *index* of the source string is less than the length of *src* but the desired length would take more characters than are left in the source string. The function simply truncates the result and returns with the carry flag clear in this situation.

This function raises an *ex.StringIndex* exception if the value of *index* is greater than the length of the *src* string. It raises an *ex.AttemptToDerefNULL* exception if *src* contains NULL. It raises an *ex.MemoryAllocation* exception if there is an error allocating sufficient storage to hold the substring. It raises an *ex.AccessViolation* if *src* contains an invalid address.

HLA high-level calling sequence examples:

```
str.a_substr( someStr, index, length );
mov( eax, subStr );
```

HLA low-level calling sequence examples:

```
push( someStr );
push( index );
```

```

push( length );
call str.a_substr;
mov( eax, subString );

```

```

procedure str.substr( src:string; index:dword; len:dword; dest:string );
    @returns( "@c" );

```

This function extracts a substring of length *len* starting at character position *index* with the source string *src*. This function stores the resulting substring into the destination string object pointed at by the *dest* argument.

If the sum of *index+len* is greater than the length of the source string, this function returns with the carry flag cleared (equal to zero) to indicate that the substring was truncated. If *index+len* is less than or equal to the length of the source string, then this function returns with the carry flag set to one. Note that this function does not raise an exception if the index of the source string is less than the length of *src* but the desired length would take more characters than are left in the source string. The function simply truncates the result and returns with the carry flag clear in this situation.

This function raises an *ex.StringIndex* exception if the value of *index* is greater than the length of the *src* string. It raises an *ex.AttemptToDerefNULL* exception if *src* or *dest* contain NULL. It raises an *ex.AccessViolation* if *src* or *dest* contain an invalid address.

**Legacy Note:** in v1.x of the HLA stdlib, the *dest* parameter was the second parameter rather than the fourth. Be aware of this issue when working with older source code.

HLA high-level calling sequence examples:

```

str.substr( someStr, index, length, subStr );

```

HLA low-level calling sequence examples:

```

push( someStr );
push( index );
push( length );
push( subStr );
call str.substr;

```

```

procedure str.a_first( src:string; len:dword );
    @returns( "@c" );

```

This function extracts a substring of length *len* starting at the beginning of the source string *src*. This function allocates storage for the substring data on the heap and returns a pointer to the new substring in the EAX register (note, however, that the function 'returns' value is "@c" and not "EAX"). It is the caller's responsibility to free the storage when it is done using the string data. This function is roughly equivalent to calling *str.a\_substr* with an index value of zero.

If the value of *len* is greater than the length of the source string, this function returns with the carry flag cleared (equal to zero) to indicate that the substring was truncated. If *len* is less than or equal to the length of the source string, then this function returns with the carry flag set to one. Note that this function does not raise an exception if the *len* argument is greater than the length of *src*. The function simply returns a copy of *src* as the result and returns with the carry flag clear in this situation.

This function an *ex.AttemptToDerefNULL* exception if *src* contains NULL. It raises an *ex.MemoryAllocation* exception if there is an error allocating sufficient storage to hold the substring. It raises an *ex.AccessViolation* if *src* contains an invalid address.

HLA high-level calling sequence examples:

```

str.a_first( someStr, length );
mov( eax, subStr );

```

HLA low-level calling sequence examples:

```
push( someStr );
push( length );
call str.a_first;
mov( eax, subString );
```

```
#macro str.first( string, dword );
#macro str.first( string, dword, dword );
```

This macro provides a "function overload" declaration for the *str.first2* and *str.first3* functions. If you pass this macro two arguments, it creates a call to the *str.first2* function; if you pass this macro three arguments, it calls the *str.first3* function.

```
procedure str.first2( src:string; len:dword );
@returns( "@c" );
```

This function extracts a substring of length *len* starting at the beginning the source string *src*. This function stores the resulting substring back into the string object pointed at by the *src* argument (that is, this function modifies the *src* argument in-place).

If the value of *len* is greater than the length of the source string, this function returns with the carry flag cleared (equal to zero) to indicate that the substring was truncated. If *len* is less than or equal to the length of the source string, then this function returns with the carry flag set to one. Note that this function does not raise an exception if the *len* argument is greater than the length of *src*. The function simply returns a copy of *src* as the result and returns with the carry flag clear in this situation.

This function raises an *ex.AttemptToDerefNULL* exception if *src* contains NULL. It raises an *ex.AccessViolation* if *src* contains an invalid address.

HLA high-level calling sequence examples:

```
str.first2( someStr, length );
```

HLA low-level calling sequence examples:

```
push( someStr );
push( length );
call str.first2;
```

```
procedure str.first3( src:string; len:dword; dest:string );
@returns( "@c" );
```

This function extracts a substring of length *len* starting at the beginning the source string *src*. This function stores the resulting substring into the destination string object pointed at by the *dest* argument.

If the value of *len* is greater than the length of the source string, this function returns with the carry flag cleared (equal to zero) to indicate that the substring was truncated. If *len* is less than or equal to the length of the source string, then this function returns with the carry flag set to one. Note that this function does not raise an exception if the *len* argument is greater than the length of *src*. The function simply returns a copy of *src* as the result and returns with the carry flag clear in this situation.

This function raises an *ex.AttemptToDerefNULL* exception if *src* or *dest* contain NULL. It raises an *ex.AccessViolation* if *src* or *dest* contain an invalid address.

HLA high-level calling sequence examples:

```
str.first3( someStr, length, subStr );
```

HLA low-level calling sequence examples:

```
push( someStr );
push( length );
push( subStr );
call str.first3;
```

```
procedure str.a_last( src:string; len:dword );
  @returns( "@c" );
```

This function extracts a substring of length *len* composed of the last *len* characters of the source string *src*. This function allocates storage for the substring data on the heap and returns a pointer to the new substring in the EAX register (note, however, that the function 'returns' value is "@c" and not "EAX"). It is the caller's responsibility to free the storage when it is done using the string data. This function is roughly equivalent to calling *str.a\_substr* with an index value of zero.

If the value of *len* is greater than the length of the source string, this function returns with the carry flag cleared (equal to zero) to indicate that the substring was shorter than expected. If *len* is less than or equal to the length of the source string, then this function returns with the carry flag set to one. Note that this function does not raise an exception if the *len* argument is greater than the length of *src*. The function simply returns a copy of *src* as the result and returns with the carry flag clear in this situation.

This function an *ex.AttemptToDerefNULL* exception if *src* contains NULL. It raises an *ex.MemoryAllocation* exception if there is an error allocating sufficient storage to hold the substring. It raises an *ex.AccessViolation* if *src* contains an invalid address.

HLA high-level calling sequence examples:

```
str.a_last( someStr, length );
mov( eax, subStr );
```

HLA low-level calling sequence examples:

```
push( someStr );
push( length );
call str.a_last;
mov( eax, subString );
```

```
#macro str.last( string, dword );
#macro str.last( string, dword, dword );
```

This macro provides a "function overload" declaration for the *str.last2* and *str.last3* functions. If you pass this macro two arguments, it creates a call to the *str.last2* function; if you pass this macro three arguments, it calls the *str.last3* function.

```
procedure str.last2( src:string; len:dword );
  @returns( "@c" );
```

This function extracts a substring of length *len* composed of the *len* characters at the end of the source string *src*. This function stores the resulting substring back into the string object pointed at by the *src* argument (that is, this function modifies the *src* argument in-place).

If the value of *len* is greater than the length of the source string, this function returns with the carry flag cleared (equal to zero) to indicate that the substring was truncated. If *len* is less than or equal to the length of the source string, then this function returns with the carry flag set to one. Note that this function does not raise an exception if the *len* argument is greater than the length of *src*. The function simply returns a copy of *src* as the result and returns with the carry flag clear in this situation.

This function raises an *ex.AttemptToDerefNULL* exception if *src* contains NULL. It raises an *ex.AccessViolation* if *src* contains an invalid address.

HLA high-level calling sequence examples:

```
str.last2( someStr, length );
```

HLA low-level calling sequence examples:

```
push( someStr );
push( length );
call str.last2;
```

```
procedure str.last3( src:string; len:dword; dest:string );
@returns( "@c" );
```

This function extracts a substring of length *len* composed of the *len* characters at the end of the source string *src*. This function stores the resulting substring into the destination string object pointed at by the *dest* argument.

If the value of *len* is greater than the length of the source string, this function returns with the carry flag cleared (equal to zero) to indicate that the substring was truncated. If *len* is less than or equal to the length of the source string, then this function returns with the carry flag set to one. Note that this function does not raise an exception if the *len* argument is greater than the length of *src*. The function simply returns a copy of *src* as the result and returns with the carry flag clear in this situation.

This function raises an *ex.AttemptToDerefNULL* exception if *src* or *dest* contain NULL. It raises an *ex.AccessViolation* if *src* or *dest* contain an invalid address.

HLA high-level calling sequence examples:

```
str.last3( someStr, length, subStr );
```

HLA low-level calling sequence examples:

```
push( someStr );
push( length );
push( subStr );
call str.last3;
```

```
procedure str.a_truncate( src:string; cnt:dword );
@returns( "@c" );
```

This function is similar to *str.a\_first* insofar as it creates a substring by extracting the characters at the beginning of the *src* string. The difference is that the *cnt* argument specifies the number of characters to delete from the end of the string rather than the length of the resulting substring. It extracts a substring of length *length(src)-cnt* starting at the beginning of the source string *src*. This function allocates storage for the substring data on the heap and returns a pointer to the new substring in the EAX register (note, however, that the function 'returns' value is "@c" and not "EAX"). It is the caller's responsibility to free the storage when it is done using the string data. This function is roughly equivalent to calling *str.a\_substr* with an index value of zero.

If the value of *cnt* is greater than the length of the source string, this function returns with the carry flag cleared (equal to zero) and returns an empty string. If *cnt* is less than or equal to the length of the source string, then this function returns with the carry flag set to one. Note that this function does not raise an exception if the *cnt* argument is greater than the length of *src*. The function simply returns the empty string as the result and returns with the carry flag clear in this situation.

This function an *ex.AttemptToDerefNULL* exception if *src* contains NULL. It raises an *ex.MemoryAllocation* exception if there is an error allocating sufficient storage to hold the substring. It raises an *ex.AccessViolation* if *src* contains an invalid address.

HLA high-level calling sequence examples:

```
str.a_truncate( someStr, charsToDelete );
mov( eax, subStr );
```

HLA low-level calling sequence examples:

```
push( someStr );
push( charsToDelete );
call str.a_truncate;
mov( eax, subString );
```

```
#macro str.truncate( string, dword );
#macro str.truncate( string, dword, dword );
```

This macro provides a "function overload" declaration for the *str.truncate2* and *str.truncate3* functions. If you pass this macro two arguments, it creates a call to the *str.truncate2* function; if you pass this macro three arguments, it calls the *str.truncate3* function.

```
procedure str.truncate2( src:string; cnt:dword );
@returns( "@c" );
```

This function extracts a substring of length *length(src)-cnt* starting at the beginning the source string *src*. This function stores the resulting substring back into the string object pointed at by the *src* argument (that is, this function modifies the *src* argument in-place).

If the value of *cnt* is greater than the length of the source string, this function returns with the carry flag cleared (equal to zero) to indicate that the substring was truncated. If *cnt* is less than or equal to the length of the source string, then this function returns with the carry flag set to one. Note that this function does not raise an exception if the *cnt* argument is greater than the length of *src*. The function simply returns an empty string as the result and returns with the carry flag clear in this situation.

This function raises an *ex.AttemptToDerefNULL* exception if *src* contains NULL. It raises an *ex.AccessViolation* if *src* contains an invalid address.

HLA high-level calling sequence examples:

```
str.truncate2( someStr, chars2Delete );
```

HLA low-level calling sequence examples:

```
push( someStr );
push( chars2Delete );
call str.truncate2;
```

```
procedure str.truncate3( src:string; len:dword; dest:string );
@returns( "@c" );
```

This function extracts a substring of length *length(src)-cnt* starting at the beginning the source string *src*. This function stores the resulting substring into the string object pointed at by the *dest* argument.

If the value of *cnt* is greater than the length of the source string, this function returns with the carry flag cleared (equal to zero) to indicate that the substring was truncated. If *cnt* is less than or equal to the length of the source string, then this function returns with the carry flag set to one. Note that this function does not raise an exception if the *cnt* argument is greater than the length of *src*. The function simply returns an empty string as the result and returns with the carry flag clear in this situation.

This function raises an *ex.AttemptToDerefNULL* exception if *src* or *dest* contain NULL. It raises an *ex.AccessViolation* if *src* or *dest* contain an invalid address.

HLA high-level calling sequence examples:

```
str.truncate3( someStr, cnt, subStr );
```

HLA low-level calling sequence examples:

```
push( someStr );
push( cnt );
push( subStr );
call str.truncate3;
```

## 33.6 String Insertion and Deletion Functions

The HLA Standard Library provides routines that insert characters (and strings) into other strings, or delete portions of a string.

```
procedure str.a_insert( ins:string; start:dword; src:string );
    @returns( "(type string eax)" );
```

This function creates a new string on the heap (returning a pointer to the new string in EAX) consisting of the characters in *src* with the *ins* string inserted at position *start*. That is, the resultant string consists of the first *start* characters of *src* followed by the characters in *ins*, followed by the remaining characters in *src* (after index *start*). Note that if *start* is equal to the length of *src*, then this function appends the *ins* string to the end of the character data from the *src* string. It is the caller's responsibility to free up the storage on the heap after the caller is done with the string data.

If the value of *start* is greater than the length of the *src* string, this function raises an *ex.StringIndex* exception. This function an *ex.AttemptToDerefNULL* exception if *src* or *ins* contain NULL. It raises an *ex.MemoryAllocation* exception if there is an error allocating sufficient storage to hold the new string. It raises an *ex.AccessViolation* if *src* or *ins* contain an invalid address.

HLA high-level calling sequence examples:

```
str.a_insert( str2insert, index, someStr );
mov( eax, newStr );
```

HLA low-level calling sequence examples:

```
push( str2insert );
push( index );
push( someStr );
call str.a_insert;
mov( eax, newStr );
```

```
#macro str.insert( string, dword, string );
#macro str.insert( string, dword, string, string );
```

This macro provides a "function overload" declaration for the *str.insert3* and *str.insert4* functions. If you pass this macro three arguments, it creates a call to the *str.insert3* function; if you pass this macro four arguments, it calls the *str.insert4* function.

```
procedure str.insert3( ins:string; start:dword; dest:string );
```

This function inserts a copy of the *ins* string at index *start* in the *dest* string. If *start* is equal to the length of *dest*, then this function concatenates the character data in *ins* to the end of the *dest* string.

If the value of *start* is greater than the length of *dest*, this function raises an *ex.StringIndex* exception.



This function raises an *ex.AttemptToDerefNULL* exception if *src* or *dest* contain NULL. It raises an *ex.AccessViolation* if *src* or *dest* contain an invalid address.

HLA high-level calling sequence examples:

```
str.insert3( str2Insert, insPosition, destStr );
```

HLA low-level calling sequence examples:

```
push( str2Insert );
push( insPosition );
push( destStr );
call str.insert3;
```

#### **procedure str.insert4**

```
(
    str2Insert    :string;
    start        :dword;
    insertInto    :string;
    dest         :string
);
```

This function creates a new string, which it copies into the string object pointed at by the *dest* argument, by inserting the *str2Insert* string into the *insertInto* string at position *start*. That is, this function copies the first *start* characters from *insertInto* to *dest*, followed by the character data pointed at by *str2Insert*, followed by the remaining characters from *insertInto*.

If the value of *start* is greater than the length of the *insertInto* string, this function raises an *ex.StringIndex* exception. This function raises an *ex.AttemptToDerefNULL* exception if *str2Insert*, *insertInto*, or *dest* contain NULL. It raises an *ex.AccessViolation* if *insertInto*, *str2Insert*, or *dest* contain an invalid address. It raises an *ex.StringOverflow* exception if the string object pointed at by *dest* isn't large enough to hold the result.

HLA high-level calling sequence examples:

```
str.insert4( insStr, position, subst, destStr );
```

HLA low-level calling sequence examples:

```
push( insStr );
push( position );
push( subst );
push( destStr );
call str.insert4;
```

```
procedure str.a_delete( src:string; start:dword; len:dword );
@returns( "@c" );
```

This function creates a new string on the heap (and returns a pointer to it in EAX) containing the characters from *src* after deleting *len* characters beginning at position *start*. It is the caller's responsibility to free the storage (e.g., via *str.free*) when it is done using the string data. This function returns with the carry flag set to denote that it created a string by deleting the specified number of characters; it returns with the carry flag clear if the string is created by deleting *len* characters because the sum of *start+len* was greater than the length of the *src* string (in which case the resulting string consists of the characters in *src* from index zero through *start-1*).

This function raises an *ex.StringIndex* exception if the value of *start* is greater than the length of the *src* string. It raises an *ex.AttemptToDerefNULL* exception if *src* contains NULL. It raises an *ex.MemoryAllocation* exception if there is an error allocating sufficient storage to hold the deleteing. It raises an *ex.AccessViolation* if *src* contains an invalid address.

HLA high-level calling sequence examples:

```
str.a_delete( someStr, index, length );
mov( eax, delete );
```

HLA low-level calling sequence examples:

```
push( someStr );
push( index );
push( length );
call str.a_delete;
mov( eax, deleteing );
```

```
#macro str.delete( string, dword, dword );
#macro str.delete( string, dword, dword, string );
```

This macro provides a "function overload" declaration for the *str.delete3* and *str.delete4* functions. If you pass this macro three arguments, it creates a call to the *str.delete3* function; if you pass this macro four arguments, it calls the *str.delete4* function.

```
procedure str.delete3( dest:string; index:dword; len:dword );
@returns( "@c" );
```

This function deletes *len* characters from *dest* starting at character position *index*. This function modifies the *dest* argument in place. This function returns with the carry flag set to denote that it created a string by deleting the specified number of characters; it returns with the carry flag clear if the string is created was unable to delete *len* characters because the sum of *start+len* was greater than the length of the *dest* string (in which case the resulting string consists of the characters in *src* from index zero through *start-1*).

This function raises an *ex.StringIndex* exception if the value of *index* is greater than the length of the *src* string. It raises an *ex.AttemptToDerefNULL* exception if *src* or *dest* contain NULL. It raises an *ex.AccessViolation* if *src* or *dest* contain an invalid address.

HLA high-level calling sequence examples:

```
str.delete3( someStr, index, length);
```

HLA low-level calling sequence examples:

```
push( someStr );
push( index );
push( length );
call str.delete3;
```

```
procedure str.delete4( src:string; index:dword; len:dword; dest:string );
@returns( "@c" );
```

This function creates a new string by deleting *len* characters in *src* starting at character position *index*. This function stores the resulting string into the destination string object pointed at by the *dest* argument. This function returns with the carry flag set to denote that it created a string by deleting the specified number of characters; it returns with the carry flag clear if the string is created was unable to delete *len* characters because the sum of *start+len* was greater than the length of the *src* string (in which case the resulting string consists of the characters in *src* from index zero through *start-1*).

This function raises an *ex.StringIndex* exception if the value of *index* is greater than the length of the *src* string. It raises an *ex.AttemptToDerefNULL* exception if *src* or *dest* contain NULL. It raises an *ex.AccessViolation* if *src* or *dest* contain an invalid address. It raises an *ex.StringOverflow* exception if the string object pointed at by *dest* is not large enough to receive the new string.

HLA high-level calling sequence examples:

```
str.delete4( someStr, index, length, delete );
```

HLA low-level calling sequence examples:

```
push( someStr );
push( index );
push( length );
push( delete );
call str.delete4;
```

```
procedure str.a_delLeadingSpaces( src:string );
@returns( "(type string eax)" );
```

This function creates a new string on the heap (and returns a pointer to it in EAX) containing the characters from *src* after deleting any leading space characters from the string. It is the caller's responsibility to free the storage allocated by this function when the storage is no longer needed (e.g., by calling *str.free*).

This function raises an *ex.AttemptToDerefNULL* exception if *src* contains NULL. It raises an *ex.MemoryAllocation* exception if there is an error allocating sufficient storage to hold the resulting string. It raises an *ex.AccessViolation* if *src* contains an invalid address.

HLA high-level calling sequence examples:

```
str.a_delLeadingSpaces( someStr );
mov( eax, newStr );
.
.
.
str.free( newStr );
```

HLA low-level calling sequence examples:

```
push( someStr );
call str.a_delLeadingSpaces;
mov( eax, newStr );
.
.
.
str.free( newStr );
```

```
#macro str.delLeadingSpaces( string, dword, dword );
#macro str.delLeadingSpaces( string, dword, dword, string );
```

This macro provides a "function overload" declaration for the *str.delLeadingSpaces1* and *str.delLeadingSpaces2* functions. If you pass this macro three arguments, it creates a call to the *str.delLeadingSpaces1* function; if you pass this macro four arguments, it calls the *str.delLeadingSpaces2* function.

**procedure str.delLeadingSpaces1( dest:string );**

This function deletes all the leading space characters from the beginning of the *dest* string. This function modifies the *dest* argument in place.

This function raises an *ex.AttemptToDerefNULL* exception if *dest* contains NULL. It raises an *ex.AccessViolation* if *dest* contains an invalid address.

HLA high-level calling sequence examples:

```
str.delLeadingSpaces1( someStr );
```

HLA low-level calling sequence examples:

```
push( someStr );
call str.delLeadingSpaces1;
```

**procedure str.delLeadingSpaces2( src:string; dest:string );**

This function creates a new string by copying all the characters from *src* to *dest* except for any leading space characters.

This function raises an *ex.AttemptToDerefNULL* exception if *src* or *dest* contain NULL. It raises an *ex.AccessViolation* if *src* or *dest* contain an invalid address.

HLA high-level calling sequence examples:

```
str.delLeadingSpaces2( someStr, trimmedStr );
```

HLA low-level calling sequence examples:

```
push( someStr );
push( trimmedStr );
call str.delLeadingSpaces2;
```

**procedure str.a\_delTrailingSpaces( src:string );  
@returns( "(type string eax)" );**

This function creates a new string on the heap (and returns a pointer to it in EAX) containing the characters from *src* after deleting any trailing space characters from the end of the string. It is the caller's responsibility to free the storage allocated by this function when the storage is no longer needed (e.g., by calling *str.free*).

This function raises an *ex.AttemptToDerefNULL* exception if *src* contains NULL. It raises an *ex.MemoryAllocation* exception if there is an error allocating sufficient storage to hold the result. It raises an *ex.AccessViolation* if *src* contains an invalid address.

HLA high-level calling sequence examples:

```
str.a_delTrailingSpaces( someStr );
mov( eax, newStr );
.
.
.
str.free( newStr );
```

HLA low-level calling sequence examples:

```

push( someStr );
call str.a_delTrailingSpaces;
mov( eax, newStr );
.
.
.
str.free( newStr );

```

```

#macro str.delTrailingSpaces( string );
#macro str.delTrailingSpaces( string, string );

```

This macro provides a "function overload" declaration for the *str.delTrailingSpaces1* and *str.delTrailingSpaces2* functions. If you pass this macro one argument, it creates a call to the *str.delTrailingSpaces1* function; if you pass this macro two arguments, it calls the *str.delTrailingSpaces2* function.

```

procedure str.delTrailingSpaces1( dest:string );

```

This function deletes all the trailing space characters from the end of the *dest* string. This function modifies the *dest* argument in place.

This function raises an *ex.AttemptToDerefNULL* exception if *dest* contains NULL. It raises an *ex.AccessViolation* if *dest* contains an invalid address.

HLA high-level calling sequence examples:

```

str.delTrailingSpaces1( someStr );

```

HLA low-level calling sequence examples:

```

push( someStr );
call str.delTrailingSpaces1;

```

```

procedure str.delTrailingSpaces2( src:string; dest:string );

```

This function creates a new string by copying all the characters from *src* to *dest* except for any trailing space characters found at the end of the *src* string.

This function raises an *ex.AttemptToDerefNULL* exception if *src* or *dest* contain NULL. It raises an *ex.AccessViolation* if *src* or *dest* contain an invalid address.

HLA high-level calling sequence examples:

```

str.delTrailingSpaces2( someStr, trimmedStr );

```

HLA low-level calling sequence examples:

```

push( someStr );
push( trimmedStr );
call str.delTrailingSpaces2;

```

```

procedure str.a_trim( src:string );
    @returns( "(type string eax)" );

```

This function creates a new string on the heap (and returns a pointer to it in EAX) containing the characters from *src* after deleting any leading and trailing space characters from the *src* string. It is the caller's responsibility to free the storage allocated by this function when the storage is no longer needed (e.g., by calling *str.free*).

This function raises an *ex.AttemptToDerefNULL* exception if *src* contains NULL. It raises an *ex.MemoryAllocation* exception if there is an error allocating sufficient storage to hold the resulting string. It raises an *ex.AccessViolation* if *src* contains an invalid address.

HLA high-level calling sequence examples:

```

    str.a_trim( someStr );
    mov( eax, newStr );
    .
    .
    .
    str.free( newStr );

```

HLA low-level calling sequence examples:

```

    push( someStr );
    call str.a_trim;
    mov( eax, newStr );
    .
    .
    .
    str.free( newStr );

```

```

#macro str.trim( string );
#macro str.trim( string, string );

```

This macro provides a "function overload" declaration for the *str.trim1* and *str.trim2* functions. If you pass this macro one argument, it creates a call to the *str.trim1* function; if you pass this macro two arguments, it calls the *str.trim2* function.

```

procedure str.trim1( dest:string );

```

This function deletes all the leading and trailing space characters from the *dest* string. This function modifies the *dest* argument in place.

This function raises an *ex.AttemptToDerefNULL* exception if *dest* contains NULL. It raises an *ex.AccessViolation* if *dest* contains an invalid address.

HLA high-level calling sequence examples:

```

    str.trim1( someStr );

```

HLA low-level calling sequence examples:

```

    push( someStr );
    call str.trim1;

```

```

procedure str.trim2( src:string; dest:string );

```

This function creates a new string by copying all the characters from *src* to *dest* except for any leading and trailing space characters found at the beginning and end of the *src* string.

This function raises an *ex.AttemptToDerefNULL* exception if *src* or *dest* contain NULL. It raises an *ex.AccessViolation* if *src* or *dest* contain an invalid address.

HLA high-level calling sequence examples:

```
str.trim2( someStr, trimmedStr );
```

HLA low-level calling sequence examples:

```
push( someStr );
push( trimmedStr );
call str.trim2;
```

```
procedure str.a_rmvTrailingSpaces( src:string );
@returns( "(type string eax)" );
```

Functionally identical to *str.a\_delTrailingSpaces* except this function deletes spaces and tab characters from the end of the source string.

This function raises an *ex.AttemptToDerefNULL* exception if *src* contains NULL. It raises an *ex.MemoryAllocation* exception if there is an error allocating sufficient storage to hold the result. It raises an *ex.AccessViolation* if *src* contains an invalid address.

HLA high-level calling sequence examples:

```
str.a_rmvTrailingSpaces( someStr );
mov( eax, newStr );
.
.
.
str.free( newStr );
```

HLA low-level calling sequence examples:

```
push( someStr );
call str.a_rmvTrailingSpaces;
mov( eax, newStr );
.
.
.
str.free( newStr );
```

```
procedure str.rmvTrailingSpaces1( dest:string );
```

Functionally identical to *str.delTrailingSpaces1s* except this function deletes spaces and tab characters from the end of the *dest* string.

This function raises an *ex.AttemptToDerefNULL* exception if *dest* contains NULL. It raises an *ex.AccessViolation* if *dest* contains an invalid address.

HLA high-level calling sequence examples:

```
str.rmvTrailingSpaces1( someStr );
```

HLA low-level calling sequence examples:

```
push( someStr );
call str.rmvTrailingSpaces1;
```

```
procedure str.rmvTrailingSpaces2( src:string; dest:string );
```

Functionally identical to *str.delTrailingSpaces2* except this function deletes spaces and tab characters from the end of the string.

This function raises an *ex.AttemptToDerefNULL* exception if *src* or *dest* contain NULL. It raises an *ex.AccessViolation* if *src* or *dest* contain an invalid address.

HLA high-level calling sequence examples:

```
str.rmvTrailingSpaces2( someStr, trimmedStr );
```

HLA low-level calling sequence examples:

```
push( someStr );
push( trimmedStr );
call str.rmvTrailingSpaces2;
```

## 33.7 String Comparison Functions

The HLA Standard Library provides routines that compare two strings and return the result of the comparison. There are two sets of comparison functions – case sensitive comparisons and case insensitive comparisons.

**Legacy Note:** These functions return true/false in the carry flag (set/clear). They preserve all the other registers. The original functions in v1.x of the HLA stdlib returned the comparison result in the EAX/AL register. If you have old code that requires the result in EAX, you can easily compute the result in EAX by placing a "mov( 0, eax );" and "adc( 0, eax );" instruction pair after the call. For example:

```
str.eq( str1, str2 );
mov( 0, eax );
adc( 0, eax );
```

```
procedure str.eq( src1:string; src2:string ); @returns( "@c" );
```

This function does a case-sensitive comparison of *src1* to *src2* and returns with the carry flag set if they are equal (carry flag is clear if they are not equal).

This function raises an *ex.AttemptToDerefNULL* exception if *src1* or *src2* contain NULL. It raises an *ex.AccessViolation* if *src1* or *src2* contain an invalid address.

HLA high-level calling sequence examples:

```
if( str.eq( hw, "Hello World" ) ) then
    // do something if hw is equal to "Hello World"
endif;
```



HLA low-level calling sequence examples:

```
push( hw );
pushd( hwLiteralString );
call str.eq;
jnc hw_NE_HelloWorld

    // do something if hw is equal to "Hello World"

hw_NE_HelloWorld:
```

**procedure str.ne( src1:string; src2:string ); @returns( "@c" );**

This function does a case-sensitive comparison of *src1* to *src2* and returns with the carry flag set if they are not equal (carry flag is clear if they are equal).

This function raises an *ex.AttemptToDerefNULL* exception if *src1* or *src2* contain NULL. It raises an *ex.AccessViolation* if *src1* or *src2* contain an invalid address.

HLA high-level calling sequence examples:

```
if( str.ne( hw, "Hello World" ) ) then

    // do something if hw is not equal to "Hello World"

endif;
```

HLA low-level calling sequence examples:

```
push( hw );
pushd( hwLiteralString );
call str.ne;
jnc hw_EQ_HelloWorld

    // do something if hw is not equal to "Hello World"

hw_EQ_HelloWorld:
```

**procedure str.lt( src1:string; src2:string ); @returns( "@c" );**

This function does a case-sensitive comparison of *src1* to *src2* and returns with the carry flag set if *src1* < *src2* (carry flag is clear if *src1* >= *src2*).

This function raises an *ex.AttemptToDerefNULL* exception if *src1* or *src2* contain NULL. It raises an *ex.AccessViolation* if *src1* or *src2* contain an invalid address.

HLA high-level calling sequence examples:

```
if( str.lt( hw, "Hello World" ) ) then

    // do something if hw is less than "Hello World"

endif;
```

HLA low-level calling sequence examples:

```
push( hw );
```

```

pushd( hwLiteralString );
call str.lt;
jnc hw_NLT>HelloWorld

    // do something if hw is less than "Hello World"

hw_NLT>HelloWorld:

```

**procedure str.le( src1:string; src2:string ); @returns( "@c" );**

This function does a case-sensitive comparison of *src1* to *src2* and returns with the carry flag set if *src1* <= *src2* (carry flag is clear if *src1* > *src2*).

This function raises an *ex.AttemptToDerefNULL* exception if *src1* or *src2* contain NULL. It raises an *ex.AccessViolation* if *src1* or *src2* contain an invalid address.

HLA high-level calling sequence examples:

```

if( str.le( hw, "Hello World" ) ) then

    // do something if hw is less than or equal to "Hello World"

endif;

```

HLA low-level calling sequence examples:

```

push( hw );
pushd( hwLiteralString );
call str.le;
jnc hw_NLE>HelloWorld

    // do something if hw is less than or equal to "Hello World"

hw_NLE>HelloWorld:

```

**procedure str.gt( src1:string; src2:string ); @returns( "@c" );**

This function does a case-sensitive comparison of *src1* to *src2* and returns with the carry flag set if *src1* > *src2* (carry flag is clear if *src1* <= *src2*).

This function raises an *ex.AttemptToDerefNULL* exception if *src1* or *src2* contain NULL. It raises an *ex.AccessViolation* if *src1* or *src2* contain an invalid address.

HLA high-level calling sequence examples:

```

if( str.gt( hw, "Hello World" ) ) then

    // do something if hw is greater than "Hello World"

endif;

```

HLA low-level calling sequence examples:

```

push( hw );
pushd( hwLiteralString );
call str.gt;
jnc hw_NGT>HelloWorld

```

```

        // do something if hw is greater than "Hello World"

hw_NGT>HelloWorld:

```

**procedure str.ge( src1:string; src2:string ); @returns( "@c" );**

This function does a case-sensitive comparison of *src1* to *src2* and returns with the carry flag set if *src1* >= *src2* (carry flag is clear if *src1* < *src2*).

This function raises an *ex.AttemptToDerefNULL* exception if *src1* or *src2* contain NULL. It raises an *ex.AccessViolation* if *src1* or *src2* contain an invalid address.

HLA high-level calling sequence examples:

```

if( str.ge( hw, "Hello World" ) ) then

    // do something if hw is greater than or equal to "Hello World"

endif;

```

HLA low-level calling sequence examples:

```

push( hw );
pushd( hwLiteralString );
call str.ge;
jnc hw_NGE>HelloWorld

    // do something if hw is greter than or equal to "Hello World"

hw_NGE>HelloWorld:

```

**procedure str.ieq( src1:string; src2:string ); @returns( "@c" );**

This function does a case-insensitive comparison of *src1* to *src2* and returns with the carry flag set if they are equal (carry flag is clear if they are not equal).

This function raises an *ex.AttemptToDerefNULL* exception if *src1* or *src2* contain NULL. It raises an *ex.AccessViolation* if *src1* or *src2* contain an invalid address.

HLA high-level calling sequence examples:

```

if( str.ieq( hw, "Hello World" ) ) then

    // do something if hw is equal to "Hello World"

endif;

```

HLA low-level calling sequence examples:

```

push( hw );
pushd( hwLiteralString );
call str.ieq;
jnc hw_NE>HelloWorld

    // do something if hw is equal to "Hello World"

hw_NE>HelloWorld:

```

```
procedure str.ine( src1:string; src2:string ); @returns( "@c" );
```

This function does a case-insensitive comparison of *src1* to *src2* and returns with the carry flag set if they are not equal (carry flag is clear if they are equal).

This function raises an *ex.AttemptToDerefNULL* exception if *src1* or *src2* contain NULL. It raises an *ex.AccessViolation* if *src1* or *src2* contain an invalid address.

HLA high-level calling sequence examples:

```
if( str.ine( hw, "Hello World" ) ) then

    // do something if hw is not equal to "Hello World"

endif;
```

HLA low-level calling sequence examples:

```
push( hw );
pushd( hwLiteralString );
call str.ine;
jnc hw_EQ_HelloWorld

    // do something if hw is not equal to "Hello World"

hw_EQ_HelloWorld:
```

```
procedure str.ilt( src1:string; src2:string ); @returns( "@c" );
```

This function does a case-insensitive comparison of *src1* to *src2* and returns with the carry flag set if *src1* < *src2* (carry flag is clear if *src1* >= *src2*).

This function raises an *ex.AttemptToDerefNULL* exception if *src1* or *src2* contain NULL. It raises an *ex.AccessViolation* if *src1* or *src2* contain an invalid address.

HLA high-level calling sequence examples:

```
if( str.ilt( hw, "Hello World" ) ) then

    // do something if hw is less than "Hello World"

endif;
```

HLA low-level calling sequence examples:

```
push( hw );
pushd( hwLiteralString );
call str.ilt;
jnc hw_NLT_HelloWorld

    // do something if hw is less than "Hello World"

hw_NLT_HelloWorld:
```

**procedure str.ile( src1:string; src2:string ); @returns( "@c" );**

This function does a case-insensitive comparison of *src1* to *src2* and returns with the carry flag set if *src1* <= *src2* (carry flag is clear if *src1* > *src2*).

This function raises an *ex.AttemptToDerefNULL* exception if *src1* or *src2* contain NULL. It raises an *ex.AccessViolation* if *src1* or *src2* contain an invalid address.

HLA high-level calling sequence examples:

```
if( str.ile( hw, "Hello World" ) ) then

    // do something if hw is less than or equal to "Hello World"

endif;
```

HLA low-level calling sequence examples:

```
push( hw );
pushd( hwLiteralString );
call str.ile;
jnc hw_NLE_HelloWorld

    // do something if hw is less than or equal to "Hello World"

hw_NLE_HelloWorld:
```

**procedure str.igt( src1:string; src2:string ); @returns( "@c" );**

This function does a case-insensitive comparison of *src1* to *src2* and returns with the carry flag set if *src1* > *src2* (carry flag is clear if *src1* <= *src2*).

This function raises an *ex.AttemptToDerefNULL* exception if *src1* or *src2* contain NULL. It raises an *ex.AccessViolation* if *src1* or *src2* contain an invalid address.

HLA high-level calling sequence examples:

```
if( str.igt( hw, "Hello World" ) ) then

    // do something if hw is greater than "Hello World"

endif;
```

HLA low-level calling sequence examples:

```
push( hw );
pushd( hwLiteralString );
call str.igt;
jnc hw_NGT_HelloWorld

    // do something if hw is greater than "Hello World"

hw_NGT_HelloWorld:
```

**procedure str.ige( src1:string; src2:string ); @returns( "@c" );**

This function does a case-insensitive comparison of *src1* to *src2* and returns with the carry flag set if *src1* >= *src2* (carry flag is clear if *src1* < *src2*).

This function raises an *ex.AttemptToDerefNULL* exception if *src1* or *src2* contain NULL. It raises an *ex.AccessViolation* if *src1* or *src2* contain an invalid address.

HLA high-level calling sequence examples:

```
if( str.ige( hw, "Hello World" ) ) then

    // do something if hw is greater than or equal to "Hello World"

endif;
```

HLA low-level calling sequence examples:

```
push( hw );
pushd( hwLiteralString );
call str.ige;
jnc hw_NGE_HelloWorld

    // do something if hw is greater than or equal to "Hello World"

hw_NGE_HelloWorld:
```

## 33.8 String Searching Functions

The HLA Standard Library provides several routines that search for strings or character patterns within other strings. These functions all return their status (true=found, false=not found) in the carry flag (set=true, clear=false). Their "returns" string is "@c" so you can call these functions in an boolean expression (e.g., in an IF statement) to test the return result.

Legacy Note: many HLA stdlib v1.x versions of these routines returned the true/false status in the EAX register. If your code requires this, then you can move the carry flag into EAX immediately after a call to one of these functions using code like the following:

```
mov( 0, eax );
adc( 0, eax );
```

```
#macro str.prefix( string, string );
#macro str.prefix( string, dword, string );
```

This macro provides a "function overload" declaration for the *str.prefix2* and *str.prefix3* functions. If you pass this macro two arguments, it creates a call to the *str.prefix2* function; if you pass this macro three arguments, it calls the *str.prefix3* function.

```
procedure str.prefix2( baseStr:string; subst:string ); @returns( "@c" );
```

This function checks to see if *subst* is a prefix of *baseStr* – that is, it compares the first characters of *baseStr* against *subst* and returns true in the carry flag if all of the characters in *subst* match the characters at the beginning of the *baseStr* string. Note that this function can still return true if *baseStr* is longer than *subst*, as long as the prefix characters of *baseStr* match all the characters of *subst* this function will return true in the carry flag.

This function raises an *ex.AttemptToDerefNULL* exception if *subst* or *baseStr* contain NULL. It raises an *ex.AccessViolation* if *subst* or *baseStr* contain an invalid address.

HLA high-level calling sequence examples:

```
if( str.prefix2( hw, "Hello World" ) ) then

    // do something if hw begins with the string "Hello World"
```

```
endif;
```

HLA low-level calling sequence examples:

```
static
  hwLiteralString :string := "Hello World";
  .
  .
  .
push( hw );
push( hwLiteralString );
call str.prefix2;
jnc hwNotPrefix;

    // do something if hw begins with the string "Hello World"

hwNotPrefix:
```

```
procedure str.prefix3( baseStr:string; offset:dword; substr:string );
@returns( "@c" );
```

This function checks to see if *subst* is a prefix of *baseStr* beginning at character position *offset* in *baseStr*—that is, it compares the characters of *baseStr* starting at position *offset* against *subst* and returns true in the carry flag if all of the characters in *subst* match the corresponding characters at the beginning of the *baseStr* string.

This function raises an *ex.AttemptToDerefNULL* exception if *subst* or *baseStr* contain NULL. It raises an *ex.AccessViolation* if *subst* or *baseStr* contain an invalid address. It raises an *ex.StringIndexError* exception if *offset* is greater than the length of *baseStr*.

HLA high-level calling sequence examples:

```
if( str.prefix3( hw, 10, "Hello World" ) ) then

    // do something if the 10th character into hw
    // starts the substring "Hello World"

endif;
```

HLA low-level calling sequence examples:

```
static
  hwLiteralString :string := "Hello World";
  .
  .
  .
push( hw );
pushd( 10 );
push( hwLiteralString );
call str.prefix3;
jnc hwNotPrefix;

    // do something if the 10th character into hw
    // starts the substring "Hello World"

hwNotPrefix:
```

```
#macro str.index( string, string );
#macro str.index( string, dword, string );
```

This macro provides a "function overload" declaration for the *str.index2* and *str.index3* functions. If you pass this macro two arguments, it creates a call to the *str.index2* function; if you pass this macro three arguments, it calls the *str.index3* function.

```
procedure str.index2( baseStr:string; subst:string ); @returns( "@c" );
```

This function checks to see if *subst* is found within *baseStr*. This function returns with the carry flag set if the substring pointed at by *subst* is present within the string pointed at by *baseStr*; it returns with the carry flag clear if *subst*'s string is not found within *baseStr*. This function also returns the index of *subst* within *baseStr* in the EAX register. If *subst*'s string is a substring of *baseStr*, then this function returns the index into *baseStr* in the EAX register; it returns -1 in EAX if *subst*'s string is not a substring of *baseStr*. If there are multiple occurrences of *subst* within *baseStr*, this function locates the first one it finds by searching from the beginning of the string.

This function raises an *ex.AttemptToDerefNULL* exception if *subst* or *baseStr* contain NULL. It raises an *ex.AccessViolation* if *subst* or *baseStr* contain an invalid address.

HLA high-level calling sequence examples:

```
if( str.index2( hw, "Hello World" ) ) then

    // do something if hw contains the string "Hello World"

endif;
```

HLA low-level calling sequence examples:

```
static
    hwLiteralString :string := "Hello World";
.
.
.
push( hw );
push( hwLiteralString );
call str.index2;
jnc hwNotInStr;

    // do something if hw contains the string "Hello World"

hwNotInStr:
```

```
procedure str.index3( baseStr:string; offset:dword; subst:string );
    @returns( "@c" );
```

This function checks to see if *subst* is found within *baseStr* starting at character position *offset* within *baseStr*. This function returns with the carry flag set if the substring pointed at by *subst* is present within the string pointed at by *baseStr+offset*; it returns with the carry flag clear if *subst*'s string is not found within *baseStr+offset*. If *subst*'s string is a substring of *baseStr+offset*, then this function returns the index into *baseStr* in the EAX register; it returns -1 in EAX if *subst*'s string is found in the substring beginning at *baseStr+offset*. If there are multiple occurrences of *subst* within *baseStr*, this function locates the first one it finds by searching for *subst* starting at character position *offset* within *baseStr*.

This function raises an *ex.AttemptToDerefNULL* exception if *subst* or *baseStr* contain NULL. It raises an *ex.AccessViolation* if *subst* or *baseStr* contain an invalid address. It raises an *ex.StringIndexError* exception if *offset* is greater than the length of *baseStr*.



HLA high-level calling sequence examples:

```

    if( str.index3( hw, 10, "Hello World" ) ) then

        // do something if the "Hello World" is found in hw
        // somewhere beyond the 10th character position.

    endif;

```

HLA low-level calling sequence examples:

```

static
    hwLiteralString :string := "Hello World";
    .
    .
    .
push( hw );
pushd( 10 );
push( hwLiteralString );
call str.index3;
jnc hwNotInStr;

    // do something if the "Hello World" is found in hw
    // somewhere beyond the 10th character position.

hwNotInStr:

```

```

#macro str.iindex( string, string );
#macro str.iindex( string, dword, string );

```

This macro provides a "function overload" declaration for the *str.iindex2* and *str.iindex3* functions. If you pass this macro two arguments, it creates a call to the *str.iindex2* function; if you pass this macro three arguments, it calls the *str.iindex3* function.

```

procedure str.iindex2( baseStr:string; subst:string ); @returns( "@c" );

```

Similar in function to *str.index2* except this function does a case-insensitive search for *subst* within *baseStr*. If there are multiple occurrences of *subst* within *baseStr*, this function locates the first one it finds by searching from the beginning of the string.

This function raises an *ex.AttemptToDerefNULL* exception if *subst* or *baseStr* contain NULL. It raises an *ex.AccessViolation* if *subst* or *baseStr* contain an invalid address.

HLA high-level calling sequence examples:

```

    if( str.iindex2( hw, "Hello World" ) ) then

        // do something if hw contains the string "Hello World"
        // (or any permutation involving upper or lower case chars)

    endif;

```

HLA low-level calling sequence examples:

```

static

```

```

    hwLiteralString :string := "Hello World";
    .
    .
    .
push( hw );
push( hwLiteralString );
call str.iindex2;
jnc hwNotInStr;

    // do something if hw contains the string "Hello World"
    // (or any permutation involving upper or lower case chars)

hwNotInStr:

```

```

procedure str.iindex3( baseStr:string; offset:dword; subst:string );
  @returns( "@c" );

```

This function is similar to *str.index3* except it does a case-insensitive search for *subst* within *baseStr*. If there are multiple occurrences of *subst* within *baseStr*, this function locates the first one it finds by searching for *subst* starting at character position *offset* within *baseStr*.

This function raises an *ex.AttemptToDerefNULL* exception if *subst* or *baseStr* contain NULL. It raises an *ex.AccessViolation* if *subst* or *baseStr* contain an invalid address. It raises an *ex.StringIndexError* exception if *offset* is greater than the length of *baseStr*.

HLA high-level calling sequence examples:

```

    if( str.iindex3( hw, 10, "Hello World" ) ) then

        // do something if the "Hello World" (or any permutation
        // involving upper and lower case) is found in hw
        // somewhere beyond the 10th character position.

    endif;

```

HLA low-level calling sequence examples:

```

static
    hwLiteralString :string := "Hello World";
    .
    .
    .
push( hw );
pushd( 10 );
push( hwLiteralString );
call str.iindex3;
jnc hwNotInStr;

    // do something if the "Hello World" (or any permutation
    // involving upper and lower case) is found in hw
    // somewhere beyond the 10th character position.

hwNotInStr:

```

```
#macro str.rindex( string, string );
#macro str.rindex( string, dword, string );
```

This macro provides a "function overload" declaration for the *str.rindex2* and *str.rindex3* functions. If you pass this macro two arguments, it creates a call to the *str.rindex2* function; if you pass this macro three arguments, it calls the *str.rindex3* function.

```
procedure str.rindex2( baseStr:string; substr:string ); @returns( "@c" );
```

This function checks to see if *subst* is found within *baseStr*. This function returns with the carry flag set if the substring pointed at by *subst* is present within the string pointed at by *baseStr*; it returns with the carry flag clear if *subst*'s string is not found within *baseStr*. This function also returns the index of *subst* within *baseStr* in the EAX register. If *subst*'s string is a substring of *baseStr*, then this function returns the index into *baseStr* in the EAX register; it returns -1 in EAX if *subst*'s string is not a substring of *baseStr*. If there are multiple occurrences of *subst* within *baseStr*, this function locates the last one it finds by searching backwards from the end of *baseStr*.

This function raises an *ex.AttemptToDerefNULL* exception if *subst* or *baseStr* contain NULL. It raises an *ex.AccessViolation* if *subst* or *baseStr* contain an invalid address.

HLA high-level calling sequence examples:

```
if( str.rindex2( hw, "Hello World" ) ) then

    // do something if hw contains the string "Hello World"

endif;
```

HLA low-level calling sequence examples:

```
static
    hwLiteralString :string := "Hello World";
    .
    .
    .
push( hw );
push( hwLiteralString );
call str.rindex2;
jnc hwNotInStr;

    // do something if hw contains the string "Hello World"

hwNotInStr:
```

```
procedure str.rindex3( baseStr:string; offset:dword; subst:string );
    @returns( "@c" );
```

This function checks to see if *subst* is found within *baseStr* starting at character position *offset* within *baseStr*. This function returns with the carry flag set if the substring pointed at by *subst* is present within the string pointed at by *baseStr+offset*; it returns with the carry flag clear if *subst*'s string is not found within *baseStr+offset*. If *subst*'s string is a substring of *baseStr+offset*, then this function returns the index into *baseStr* in the EAX register; it returns -1 in EAX if *subst*'s string is found in the substring beginning at *baseStr+offset*. If there are multiple occurrences of *subst* within *baseStr*, this function locates the first one it finds by searching for *subst* starting at the last character position within *baseStr* up to character position *offset*.

This function raises an *ex.AttemptToDerefNULL* exception if *subst* or *baseStr* contain NULL. It raises an *ex.AccessViolation* if *subst* or *baseStr* contain an invalid address. It raises an *ex.StringIndexError* exception if *offset* is greater than the length of *baseStr*.

HLA high-level calling sequence examples:

```
if( str.rindex3( hw, 10, "Hello World" ) ) then
```

```

        // do something if the "Hello World" is found in hw
        // somewhere beyond the 10th character position.

    endif;

```

HLA low-level calling sequence examples:

```

static
    hwLiteralString :string := "Hello World";
    .
    .
    .
push( hw );
pushd( 10 );
push( hwLiteralString );
call str.rindex3;
jnc hwNotInStr;

        // do something if the "Hello World" is found in hw
        // somewhere beyond the 10th character position.

hwNotInStr:

```

```

#macro str.irindex( string, string );
#macro str.irindex( string, dword, string );

```

This macro provides a "function overload" declaration for the *str.irindex2* and *str.irindex3* functions. If you pass this macro two arguments, it creates a call to the *str.irindex2* function; if you pass this macro three arguments, it calls the *str.irindex3* function.

```

procedure str.irindex2( baseStr:string; subst:string ); @returns( "@c" );

```

Similar in function to *str.rindex2* except this function does a case-insensitive search for *subst* within *baseStr*. If there are multiple occurrences of *subst* within *baseStr*, this function locates the last one in *baseStr* by searching backwards from the end of the string.

This function raises an *ex.AttemptToDerefNULL* exception if *subst* or *baseStr* contain NULL. It raises an *ex.AccessViolation* if *subst* or *baseStr* contain an invalid address.

HLA high-level calling sequence examples:

```

    if( str.irindex2( hw, "Hello World" ) ) then

        // do something if hw contains the string "Hello World"
        // (or any permutation involving upper or lower case chars)

    endif;

```

HLA low-level calling sequence examples:

```

static
    hwLiteralString :string := "Hello World";
    .
    .
    .

```

```

push( hw );
push( hwLiteralString );
call str.irindex2;
jnc hwNotInStr;

    // do something if hw contains the string "Hello World"
    // (or any permutation involving upper or lower case chars)

hwNotInStr:

```

```

procedure str.irindex3( baseStr:string; offset:dword; subst:string );
    @returns( "@c" );

```

This function is similar to *str.rindex3* except it does a case-insensitive search for *subst* within *baseStr*. If there are multiple occurrences of *subst* within *baseStr*, this function locates the last one in *baseStr* by searching backwards for *subst* starting at the end of *baseStr* down to character position *offset*.

This function raises an *ex.AttemptToDerefNULL* exception if *subst* or *baseStr* contain NULL. It raises an *ex.AccessViolation* if *subst* or *baseStr* contain an invalid address. It raises an *ex.StringIndexError* exception if *offset* is greater than the length of *baseStr*.

HLA high-level calling sequence examples:

```

if( str.irindex3( hw, 10, "Hello World" ) ) then

    // do something if the "Hello World" (or any permutation
    // involving upper and lower case) is found in hw
    // somewhere beyond the 10th character position.

endif;

```

HLA low-level calling sequence examples:

```

static
    hwLiteralString :string := "Hello World";
    .
    .
    .
push( hw );
pushd( 10 );
push( hwLiteralString );
call str.irindex3;
jnc hwNotInStr;

    // do something if the "Hello World" (or any permutation
    // involving upper and lower case) is found in hw
    // somewhere beyond the 10th character position.

hwNotInStr:

```

```
#macro str.chpos( string, string );
#macro str.chpos( string, dword, string );
```

This macro provides a "function overload" declaration for the *str.chpos2* and *str.chpos3* functions. If you pass this macro two arguments, it creates a call to the *str.chpos2* function; if you pass this macro three arguments, it calls the *str.chpos3* function.

```
procedure str.chpos2( baseStr:string; src:char ); @returns( "@c" );
```

This function checks to see if *src* is found within *baseStr*. This function returns with the carry flag set if the character *src* is present within the string pointed at by *baseStr*; it returns with the carry flag clear if *src* is not found within *baseStr*. This function also returns the index of *src* within *baseStr* in the EAX register. If *src* is present in *baseStr*, then this function returns the index into *baseStr* in the EAX register; it returns -1 in EAX if *src* is not present in *baseStr*. If there are multiple occurrences of *src* within *baseStr*, this function locates the first one it finds by searching from the beginning of the string.

This function raises an *ex.AttemptToDerefNULL* exception if *baseStr* contains NULL. It raises an *ex.AccessViolation* if *baseStr* contains an invalid address.

HLA high-level calling sequence examples:

```
if( str.chpos2( someStr, 'a' ) ) then

    // do something if hw contains the character 'a'

endif;
```

HLA low-level calling sequence examples:

```
push( someStr );
pushd( 'a' );
call str.chpos2;
jnc aNotInStr;

    // do something if hw contains the character 'a'.

aNotInStr:
```

```
procedure str.chpos3( baseStr:string; offset:dword; src:char );
@returns( "@c" );
```

This function checks to see if *src* is found within *baseStr* starting at character position *offset* within *baseStr*. This function returns with the carry flag set if *src* is present within the string pointed at by *baseStr+offset*; it returns with the carry flag clear if *src* is not found within *baseStr+offset*. If *src* is found in *baseStr+offset*, then this function returns the index into *baseStr* in the EAX register; it returns -1 in EAX if *src* is found in the substring beginning at *baseStr+offset*. If there are multiple occurrences of *src* within *baseStr*, this function locates the first one it finds by searching for *src* starting at character position *offset* within *baseStr*.

This function raises an *ex.AttemptToDerefNULL* exception if *subst* or *baseStr* contain NULL. It raises an *ex.AccessViolation* if *subst* or *baseStr* contain an invalid address. It raises an *ex.StringIndexError* exception if *offset* is greater than the length of *baseStr*.

HLA high-level calling sequence examples:

```
if( str.chpos3( hw, 10, 'a' ) ) then

    // do something if the 'a' is found in hw
    // somewhere beyond the 10th character position.

endif;
```

HLA low-level calling sequence examples:

```

push( hw );
pushd( 10 );
pushd( 'a' );
call str.chpos3;
jnc hwNotInStr;

    // do something if the 'a' is found in hw
    // somewhere beyond the 10th character position.

hwNotInStr:

```

```

#macro str.ichpos( string, string );
#macro str.ichpos( string, dword, string );

```

This macro provides a "function overload" declaration for the *str.ichpos2* and *str.ichpos3* functions. If you pass this macro two arguments, it creates a call to the *str.ichpos2* function; if you pass this macro three arguments, it calls the *str.ichpos3* function.

```

procedure str.ichpos2( baseStr:string; src:char ); @returns( "@c" );

```

Similar in function to *str.chpos2* except this function does a case-insensitive search for *src* within *baseStr*. If there are multiple occurrences of *src* within *baseStr*, this function locates the first one it finds by searching from the beginning of the string.

This function raises an *ex.AttemptToDerefNULL* exception if *subst* or *baseStr* contain NULL. It raises an *ex.AccessViolation* if *subst* or *baseStr* contain an invalid address.

HLA high-level calling sequence examples:

```

if( str.ichpos2( someStr, 'b' ) ) then

    // do something if someStr contains the character 'b' or 'B'

endif;

```

HLA low-level calling sequence examples:

```

push( someStr );
pushd( 'b' );
call str.ichpos2;
jnc hwNotInStr;

    // do something if someStr contains the character 'b' or 'B'

hwNotInStr:

```

```

procedure str.ichpos3( baseStr:string; offset:dword; src:char );
@returns( "@c" );

```

This function is similar to *str.chpos3* except it does a case-insensitive search for *src* within *baseStr*. If there are multiple occurrences of *src* within *baseStr*, this function locates the first one it finds by searching for *src* starting at character position *offset* within *baseStr*.

This function raises an *ex.AttemptToDerefNULL* exception if *subst* or *baseStr* contain NULL. It raises an *ex.AccessViolation* if *subst* or *baseStr* contain an invalid address. It raises an *ex.StringIndexError* exception if *offset* is greater than the length of *baseStr*.

HLA high-level calling sequence examples:

```
if( str.ichpos3( someStr, 10, 'c' ) ) then

    // do something if 'c' or 'C' is found in someStr
    // somewhere beyond the 10th character position.

endif;
```

HLA low-level calling sequence examples:

```
push( someStr );
pushd( 10 );
pushd( 'c' );
call str.ichpos3;
jnc hwNotInStr;

    // do something if 'c' or 'C' is found in someStr
    // somewhere beyond the 10th character position.

hwNotInStr:
```

```
#macro str.rchpos( string, string );
#macro str.rchpos( string, dword, string );
```

This macro provides a "function overload" declaration for the *str.rchpos2* and *str.rchpos3* functions. If you pass this macro two arguments, it creates a call to the *str.rchpos2* function; if you pass this macro three arguments, it calls the *str.rchpos3* function.

```
procedure str.rchpos2( baseStr:string; src:char ); @returns( "@c" );
```

This function checks to see if *src* is found within *baseStr*. This function returns with the carry flag set if *src* is present within the string pointed at by *baseStr*; it returns with the carry flag clear if *src* is not found within *baseStr*. This function also returns the index of *src* within *baseStr* in the EAX register. If *src* is in *baseStr*, then this function returns the index into *baseStr* in the EAX register; it returns -1 in EAX if *src* is not present in *baseStr*. If there are multiple occurrences of *src* within *baseStr*, this function locates the last one it finds by searching backwards from the end of *baseStr*.

This function raises an *ex.AttemptToDerefNULL* exception if *subst* or *baseStr* contain NULL. It raises an *ex.AccessViolation* if *subst* or *baseStr* contain an invalid address.

HLA high-level calling sequence examples:

```
if( str.rchpos2( someStr, 'd' ) ) then

    // do something if someStr contains 'd'

endif;
```

HLA low-level calling sequence examples:

```
static
```



```

push( someStr );
pushd( 'd' );
call str.rchpos2;
jnc hwNotInStr;

    // do something if someStr contains 'd'

hwNotInStr:

```

```

procedure str.rchpos3( baseStr:string; offset:dword; src:char );
    @returns( "@c" );

```

This function checks to see if *src* is found within *baseStr* starting at character position *offset* within *baseStr*. This function returns with the carry flag set if *src* is present within the string pointed at by *baseStr+offset*; it returns with the carry flag clear if *src* is not found within *baseStr+offset*. If *src* is present in *baseStr+offset*, then this function returns the index into *baseStr* in the EAX register; it returns -1 in EAX if *src* is not found in the substring beginning at *baseStr+offset*. If there are multiple occurrences of *src* within *baseStr*, this function locates the first one it finds by searching backwards for *src* starting at the last character position within *baseStr* (down to character position *offset*).

This function raises an *ex.AttemptToDerefNULL* exception if *subst* or *baseStr* contain NULL. It raises an *ex.AccessViolation* if *subst* or *baseStr* contain an invalid address. It raises an *ex.StringIndexError* exception if *offset* is greater than the length of *baseStr*.

HLA high-level calling sequence examples:

```

if( str.rchpos3( someStr, 10, 'e' ) ) then

    // do something if 'e' is found in someStr
    // somewhere beyond the 10th character position.

endif;

```

HLA low-level calling sequence examples:

```

static
    hwLiteralString :string := "Hello World";
    .
    .
    .
push( someStr );
pushd( 10 );
pushd( 'e' );
call str.rchpos3;
jnc hwNotInStr;

    // do something if 'e' is found in someStr
    // somewhere beyond the 10th character position.

hwNotInStr:

```

```

#macro str.irchpos( string, string );
#macro str.irchpos( string, dword, string );

```

This macro provides a "function overload" declaration for the *str.irchpos2* and *str.irchpos3* functions. If you pass this macro two arguments, it creates a call to the *str.irchpos2* function; if you pass this macro three arguments, it calls the *str.irchpos3* function.

```
procedure str.irchpos2( baseStr:string; src:char ); @returns( "@c" );
```

Similar in function to *str.rchpos2* except this function does a case-insensitive search for *src* within *baseStr*. If there are multiple occurrences of *src* within *baseStr*, this function locates the last one in *baseStr* by searching backwards from the end of the string.

This function raises an *ex.AttemptToDerefNULL* exception if *subst* or *baseStr* contain NULL. It raises an *ex.AccessViolation* if *subst* or *baseStr* contain an invalid address.

HLA high-level calling sequence examples:

```
if( str.irchpos2( someStr, 'f' ) ) then
    // do something if someStr contains 'f' or 'F'
endif;
```

HLA low-level calling sequence examples:

```
push( someStr );
pushd( 'f' );
call str.irchpos2;
jnc hwNotInStr;

    // do something if someStr contains 'f' or 'F'

hwNotInStr:
```

```
procedure str.irchpos3( baseStr:string; offset:dword; src:char );
    @returns( "@c" );
```

This function is similar to *str.rchpos3* except it does a case-insensitive search for *src* within *baseStr*. If there are multiple occurrences of *src* within *baseStr*, this function locates the last one in *baseStr* by searching backwards for *src* starting at the end of *baseStr* down to character position *offset*.

This function raises an *ex.AttemptToDerefNULL* exception if *subst* or *baseStr* contain NULL. It raises an *ex.AccessViolation* if *subst* or *baseStr* contain an invalid address. It raises an *ex.StringIndexError* exception if *offset* is greater than the length of *baseStr*.

HLA high-level calling sequence examples:

```
if( str.irchpos3( someStr, 10, 'g' ) ) then
    // do something if 'g' or 'G' is found in someStr
    // somewhere beyond the 10th character position.
endif;
```

HLA low-level calling sequence examples:

```
push( someStr );
pushd( 10 );
pushd( 'g' );
call str.irchpos3;
jnc hwNotInStr;

    // do something if 'g' or 'G' is found in someStr
```

```
// somewhere beyond the 10th character position.

hwNotInStr:
```

## 33.9 Character Set Searching Functions

The HLA Standard Library provides several routines that test characters in strings to see if they are members of some character set. . These functions all return their status (true=found, false=not found) in the carry flag (set=true, clear=false). Their "returns" string is "@c" so you can call these functions in an boolean expression (e.g., in an IF statement) to test the return result.

```
#macro str.span( string, cset );
#macro str.span( string, dword, cset );
```

The *str.span* macro overloads the *str.span2* and *str.span3* procedures. The *str.span* macro is deprecated. New code should use the *str.skipInCset* macro instead.

```
procedure str.span2( baseStr:string; src:cset );
@returns( "eax" );
```

This is a deprecated, legacy, function from the HLA stdlib v1.x package. This function has been renamed *str.skipInCset2*. See that function's description for details.

```
procedure str.span3( baseStr:string; offset:dword; src:cset );
@returns( "eax" );
```

This is a deprecated, legacy, function from the HLA stdlib v1.x package. This function has been renamed *str.skipInCset3*. See that function's description for details.

```
#macro str.rspan( string, cset );
#macro str.rspan( string, dword, cset );
```

The *str.rspan* macro overloads the *str.span2* and *str.span3* procedures. The *str.rspan* macro is deprecated. New code should use the *str.skipInCset* macro instead.

```
procedure str.rspan2( baseStr:string; src:cset );
@returns( "eax" );
```

This is a deprecated, legacy, function from the HLA stdlib v1.x package. This function has been renamed *str.rskipInCset2*. See that function's description for details.

```
procedure str.rspan3( baseStr:string; offset:dword; src:cset );
@returns( "eax" );
```

This is a deprecated, legacy, function from the HLA stdlib v1.x package. This function has been renamed *str.rskipInCset3*. See that function's description for details.

```
#macro str.brk( string, cset );
#macro str.brk( string, dword, cset );
```

The *str.brk* macro overloads the *str.brk2* and *str.brk3* procedures. The *str.brk* macro is deprecated. New code should use the *str.findInCset* macro instead.

```
procedure str.brk2( baseStr:string; src:cset );
@returns( "eax" );
```

This is a deprecated, legacy, function from the HLA stdlib v1.x package. This function has been renamed *str.findInCset2*. See that function's description for details.

```

procedure str.brk3( baseStr:string; offset:dword; src:cset );
    @returns( "eax" );

```

This is a deprecated, legacy, function from the HLA stdlib v1.x package. This function has been renamed *str.findInCset3*. See that function's description for details.

```

#macro str.rbrk( string, cset );
#macro str.rbrk( string, dword, cset );

```

The *str.rbrk* macro overloads the *str.brk2* and *str.brk3* procedures. The *str.brk* macro is deprecated. New code should use the *str.findInCset* macro instead.

```

procedure str.rbrk2( baseStr:string; src:cset );
    @returns( "eax" );

```

This is a deprecated, legacy, function from the HLA stdlib v1.x package. This function has been renamed *str.rfindInCset2*. See that function's description for details.

```

procedure str.rbrk3( baseStr:string; offset:dword; src:cset );
    @returns( "eax" );

```

This is a deprecated, legacy, function from the HLA stdlib v1.x package. This function has been renamed *str.rfindInCset3*. See that function's description for details.

```

#macro str.skipInCset( string, cset );
#macro str.skipInCset( string, dword, cset );

```

This macro provides a "function overload" declaration for the *str.skipInCset2* and *str.skipInCset3* functions. If you pass this macro two arguments, it creates a call to the *str.skipInCset2* function; if you pass this macro three arguments, it calls the *str.skipInCset3* function.

```

procedure str.skipInCset2( baseStr:string; src:cset ); @returns( "@c" );

```

This function scans over characters in *baseStr* that are members of the *src* character set. It returns with the carry flag set if there is at least one character at the beginning of *src* that is a member of the *src* cset; it returns with the carry flag clear if the first character of *baseStr* is not a member of *src*. This function also returns the index of the first character in *baseStr* that is not a member of *src* in the EAX register; it returns -1 in EAX if the first character of *baseStr* is not a member of *src*.

This function raises an *ex.AttemptToDerefNULL* exception if *baseStr* contains NULL. It raises an *ex.AccessViolation* if *baseStr* contains an invalid address.

HLA high-level calling sequence examples:

```

    if( str.skipInCset2( someStr, chars.AlphaChars ) ) then

        // EAX contains the index of the first non-alphabetic
        // character in someStr at this point.

    endif;

```

HLA low-level calling sequence examples:

```

push( someStr );
push( (type dword chars.AlphaChars[12]) );
push( (type dword chars.AlphaChars[ 8]) );
push( (type dword chars.AlphaChars[ 4]) );
push( (type dword chars.AlphaChars[ 0]) );
call str.skipInCset2;
jnc ssNotInSet;

```

```
// EAX contains the index of the first non-alphabetic
// character in someStr at this point.
```

```
ssNotInSet:
```

```
procedure str.skipInCset3( baseStr:string; offset:dword; src:cset );
    @returns( "@c" );
```

This function scans over characters in *baseStr*, starting at character position *offset*, that are members of the *src* character set. It returns with the carry flag set if there is at least one character that is a member of the *src* cset; it returns with the carry flag clear if the first character matched is not a member of *src*. This function also returns the index of the first character in *baseStr* that is not a member of *src* in the EAX register; it returns -1 in EAX if the first character it tests is not a member of *src*. Note that the value returned is the index from the beginning of the string, not from the *offset* position in the string.

This function raises an *ex.AttemptToDerefNULL* exception if *baseStr* contains NULL. It raises an *ex.AccessViolation* if *baseStr* contains an invalid address.

HLA high-level calling sequence examples:

```
if( str.skipInCset3( someStr, 10, chars.AlphaChars ) ) then

    // EAX contains the index of the first non-alphabetic
    // character starting at position 10 in someStr.

endif;
```

HLA low-level calling sequence examples:

```
push( someStr );
pushd( 10 );
push( (type dword chars.AlphaChars[12]) );
push( (type dword chars.AlphaChars[ 8]) );
push( (type dword chars.AlphaChars[ 4]) );
push( (type dword chars.AlphaChars[ 0]) );
call str.skipInCset3;
jnc ssNotInSet;

// EAX contains the index of the first non-alphabetic
// character starting at position 10 in someStr.

ssNotInSet:
```

```
#macro str.rskipInCset( string, cset );
#macro str.rskipInCset( string, dword, cset );
```

This macro provides a "function overload" declaration for the *str.rskipInCset2* and *str.rskipInCset3* functions. If you pass this macro two arguments, it creates a call to the *str.rskipInCset2* function; if you pass this macro three arguments, it calls the *str.rskipInCset3* function.

```
procedure str.rskipInCset2( baseStr:string; src:cset ); @returns( "@c" );
```

This function scans over characters in *baseStr* that are members of the *src* character set. It returns with the carry flag set if there is at least one character in *src*, scanning from the end of *src*, that is a member of the *src* cset; it returns with the carry flag clear if the last character of *baseStr* is not a member of *src*. This function also returns the index of the last character in *baseStr* that is not a member of *src* (scanning from the end of *src*) in the EAX register; it returns -1 in EAX if the first character of *baseStr* is not a member of *src*.

This function raises an *ex.AttemptToDerefNULL* exception if *baseStr* contains NULL. It raises an *ex.AccessViolation* if *baseStr* contains an invalid address.

HLA high-level calling sequence examples:

```
if( str.rskipInCset2( someStr, chars.AlphaChars ) ) then

    // EAX contains the index of the last non-alphabetic
    // character in someStr at this point.

endif;
```

HLA low-level calling sequence examples:

```
push( someStr );
push( (type dword chars.AlphaChars[12]) );
push( (type dword chars.AlphaChars[ 8]) );
push( (type dword chars.AlphaChars[ 4]) );
push( (type dword chars.AlphaChars[ 0]) );
call str.rskipInCset2;
jnc ssNotInSet;

    // EAX contains the index of the last non-alphabetic
    // character in someStr at this point.
```

ssNotInSet:

```
procedure str.rskipInCset3( baseStr:string; offset:dword; src:cset );
@returns( "@c" );
```

This function scans over characters in *baseStr*, starting at the end of *src* and working down to character position *offset*, that are members of the *src* character set. It returns with the carry flag set if there is at least one character that is a member of the *src* cset; it returns with the carry flag clear if the last character in *src* is not a member of *src*. This function also returns the index of the last character in *baseStr* that is not a member of *src* in the EAX register; it returns -1 in EAX if the first character it tests is not a member of *src*. Note that the value returned is the index from the beginning of the string, not from the *offset* position in the string.

This function raises an *ex.AttemptToDerefNULL* exception if *baseStr* contains NULL. It raises an *ex.AccessViolation* if *baseStr* contains an invalid address.

HLA high-level calling sequence examples:

```
if( str.rskipInCset3( someStr, 10, chars.AlphaChars ) ) then

    // EAX contains the index of the last non-alphabetic
    // character down to position 10 in someStr.

endif;
```

HLA low-level calling sequence examples:

```
push( someStr );
pushd( 10 );
push( (type dword chars.AlphaChars[12]) );
push( (type dword chars.AlphaChars[ 8]) );
push( (type dword chars.AlphaChars[ 4]) );
push( (type dword chars.AlphaChars[ 0]) );
call str.rskipInCset3;
jnc ssNotInSet;

    // EAX contains the index of the last non-alphabetic
    // character down to position 10 in someStr.
```

```
ssNotInSet:
```

```
#macro str.findInCset( string, cset );
#macro str.findInCset( string, dword, cset );
```

This macro provides a "function overload" declaration for the *str.findInCset2* and *str.findInCset3* functions. If you pass this macro two arguments, it creates a call to the *str.findInCset2* function; if you pass this macro three arguments, it calls the *str.findInCset3* function.

```
procedure str.findInCset2( baseStr:string; src:cset ); @returns( "@c" );
```

This function scans over characters in *baseStr* that are not members of the *src* character set. It returns with the carry flag set if there is at least one character in *src* that is a member of the *src* cset; it returns with the carry flag clear if all the characters of *baseStr* are not members of *src*. This function also returns the index of the first character in *baseStr* that is a member of *src* in the EAX register; it returns -1 in EAX if all the characters of *baseStr* are not members of *src*.

This function raises an *ex.AttemptToDerefNULL* exception if *baseStr* contains NULL. It raises an *ex.AccessViolation* if *baseStr* contains an invalid address.

HLA high-level calling sequence examples:

```
if( str.findInCset2( someStr, chars.AlphaChars ) ) then

    // EAX contains the index of the first alphabetic
    // character in someStr at this point.

endif;
```

HLA low-level calling sequence examples:

```
push( someStr );
push( (type dword chars.AlphaChars[12]) );
push( (type dword chars.AlphaChars[ 8]) );
push( (type dword chars.AlphaChars[ 4]) );
push( (type dword chars.AlphaChars[ 0]) );
call str.findInCset2;
jnc ssNotInSet;

    // EAX contains the index of the first alphabetic
    // character in someStr at this point.
```

```
ssNotInSet:
```

```
procedure str.findInCset3( baseStr:string; offset:dword; src:cset );
    @returns( "@c" );
```

This function scans over characters in *baseStr*, starting at character position *offset*, that are not members of the *src* character set. It returns with the carry flag set if there is at least one character that is a member of the *src* cset; it returns with the carry flag clear if all the characters are not members of *src*. This function also returns the index of the first character in *baseStr* that is a member of *src* in the EAX register; it returns -1 in EAX if all the characters it tests are not members of *src*. Note that the value returned is the index from the beginning of the string, not from the *offset* position in the string.

This function raises an *ex.AttemptToDerefNULL* exception if *baseStr* contains NULL. It raises an *ex.AccessViolation* if *baseStr* contains an invalid address.

HLA high-level calling sequence examples:

```
if( str.findInCset3( someStr, 10, chars.AlphaChars ) ) then
```

```

        // EAX contains the index of the first alphabetic
        // character starting at position 10 in someStr.

    endif;

HLA low-level calling sequence examples:

push( someStr );
pushd( 10 );
push( (type dword chars.AlphaChars[12]) );
push( (type dword chars.AlphaChars[ 8]) );
push( (type dword chars.AlphaChars[ 4]) );
push( (type dword chars.AlphaChars[ 0]) );
call str.findInCset3;
jnc ssNotInSet;

        // EAX contains the index of the first alphabetic
        // character starting at position 10 in someStr.

ssNotInSet:

```

```

#macro str.rfindInCset( string, cset );
#macro str.rfindInCset( string, dword, cset );

```

This macro provides a "function overload" declaration for the *str.rfindInCset2* and *str.rfindInCset3* functions. If you pass this macro two arguments, it creates a call to the *str.rfindInCset2* function; if you pass this macro three arguments, it calls the *str.rfindInCset3* function.

```

procedure str.rfindInCset2( baseStr:string; src:cset ); @returns( "@c" );

```

This function scans over characters in *baseStr* that are not members of the *src* character set. It returns with the carry flag set if there is at least one character in *src*, scanning from the end of *src*, that is a member of the *src* cset; it returns with the carry flag clear if none of the characters of *baseStr* are members of *src*. This function also returns the index of the last character in *baseStr* that is a member of *src* (scanning from the end of *src*) in the EAX register; it returns -1 in EAX if none of the characters of *baseStr* are members of *src*.

This function raises an *ex.AttemptToDerefNULL* exception if *baseStr* contains NULL. It raises an *ex.AccessViolation* if *baseStr* contains an invalid address.

```

HLA high-level calling sequence examples:

    if( str.rfindInCset2( someStr, chars.AlphaChars ) ) then

        // EAX contains the index of the last alphabetic
        // character in someStr at this point.

    endif;

```

HLA low-level calling sequence examples:

```

push( someStr );
push( (type dword chars.AlphaChars[12]) );
push( (type dword chars.AlphaChars[ 8]) );
push( (type dword chars.AlphaChars[ 4]) );
push( (type dword chars.AlphaChars[ 0]) );
call str.rfindInCset2;
jnc ssNotInSet;

```



```

        // EAX contains the index of the last alphabetic
        // character in someStr at this point.

ssNotInSet:

```

```

procedure str.rfindInCset3( baseStr:string; offset:dword; src:cset );
    @returns( "@c" );

```

This function scans over characters in *baseStr*, starting at the end of *src* and working down to character position *offset*, that are not members of the *src* character set. It returns with the carry flag set if there is at least one character that is a member of the *src* cset; it returns with the carry flag clear if none of the characters in *src* are members of *src*. This function also returns the index of the last character in *baseStr* that is a member of *src* in the EAX register; it returns -1 in EAX if none of the characters it tests are members of *src*. Note that the value returned is the index from the beginning of the string, not from the *offset* position in the string.

This function raises an *ex.AttemptToDerefNULL* exception if *baseStr* contains NULL. It raises an *ex.AccessViolation* if *baseStr* contains an invalid address.

HLA high-level calling sequence examples:

```

    if( str.rfindInCset3( someStr, 10, chars.AlphaChars ) ) then

        // EAX contains the index of the last alphabetic
        // character down to position 10 in someStr.

    endif;

```

HLA low-level calling sequence examples:

```

push( someStr );
pushd( 10 );
push( (type dword chars.AlphaChars[12]) );
push( (type dword chars.AlphaChars[ 8]) );
push( (type dword chars.AlphaChars[ 4]) );
push( (type dword chars.AlphaChars[ 0]) );
call str.rfindInCset3;
jnc ssNotInSet;

    // EAX contains the index of the last alphabetic
    // character down to position 10 in someStr.

ssNotInSet:

```

## 33.10 String Parsing Functions

The HLA Standard Library provides several routines allow you to deconstruct and reconstruct string objects. These functions separate strings into an array of words (tokens) or otherwise break up the string in pieces by extracting portions of the string.

```

procedure str.tokenCnt1( src:string ); @returns( "eax" );

```

This function counts the number of tokens, or words, present in the *src* string. A token is considered to be any text separated by members from the *str.CmdLnDelimiters* character set ( { #0, ' ', #9, '<', '>', '|', '\', '/', '-' } ) or the beginning or end of the string.

This function raises an *ex.AttemptToDerefNULL* exception if *src* contains NULL. It raises an *ex.AccessViolation* if *src* contains an invalid address.

HLA high-level calling sequence examples:

```
str.tokenCnt1( someStr );
mov( eax, numTokens );
```

HLA low-level calling sequence examples:

```
push( someStr );
call str.tokenCnt1;
mov( eax, numTokens );
```

**procedure str.tokenCnt2( src:string; delimiters:cset ); @returns( "eax" );**

This function counts the number of tokens, or words, present in the *src* string. A token is considered to be any text separated by members from the *delimiters* character set or the beginning or end of the string.

This function raises an *ex.AttemptToDerefNULL* exception if *src* contains NULL. It raises an *ex.AccessViolation* if *src* contains an invalid address.

HLA high-level calling sequence examples:

```
str.tokenCnt2( someStr, chars.WhiteSpaceCset );
mov( eax, numTokens );
```

HLA low-level calling sequence examples:

```
push( someStr );
push( (type dword chars.WhiteSpaceCset[12]) );
push( (type dword chars.WhiteSpaceCset[ 8]) );
push( (type dword chars.WhiteSpaceCset[ 4]) );
push( (type dword chars.WhiteSpaceCset[ 0]) );
call str.tokenCnt2;
mov( eax, numTokens );
```

**procedure str.tokenize( src:string; delimiters:cset ); @returns( "eax" );**

This function is an alias for *str.tokenize3* maintained for legacy purposes. New code should use the *str.tokenize3* name.

**procedure str.tokenize3( src:string; var dest:var; maxStrs:dword );  
@returns( "eax" );**

This function lexically scans the *src* string and breaks it up into an array of strings with each element of the array containing one "token" string. This function uses the *str.CmdLnDelimiters* character set ( { #0, ',', #9, '!', '<', '>', '|', '\', '/', '-' } ) to delimit the lexemes it produces. This character set roughly corresponds to the delimiters used by the Windows Command Window interpreter or typical Linux shells. If you do not wish to use this particular set of delimiter characters, you may call *str.tokenize4* and specify the characters you're interested in.

The *str.tokenize3* routine begins by skipping over all delimiter characters at the beginning of the string. Once it locates a non-delimiter character, it skips forward until it finds the end of the string or the next delimiter character. It then allocates storage for a new string on the heap and copies the delimited text to this new string. A pointer to the new string is stored into the *dest* array passed as the second parameter. This process is repeated for each lexeme found in the *src* string.

As this function is intended for processing command lines, any quoted string (a sequence of characters surrounded by quotes or apostrophies) is treated as a single token/string by these functions. If this behavior is a problem for you, it's real easy to modify the *str.tokenize3* source file to handle this issue.

**Warning:** the *dest* parameter must be an array of string pointers. This array must be large enough to hold pointers to each lexeme found in the string. In theory, there could be as many as *str.length(src)/2* lexemes in the source string. The *maxStrs* parameter specifies the maximum number of strings this function can store into the array pointed at by *dest*.

On return from these functions, the EAX register will contain the number of lexemes found and processed in the *src* string (i.e., EAX will contain the number of valid elements in the *dest* array).

When you are done with the strings allocated on the heap, you should free them by calling *str.free*. Note that you need to call *str.free* for each active pointer stored in the *dest* array.

Here is an example of a call to the *str.tokenize3* routine:

```
program tokenizeDemo;
#include( "stdlib.hhf" );

static
  strings: string[16];
  ParseMe: string := "This string contains five words";

begin tokenizeDemo;

  str.tokenize3( ParseMe, strings, 16 );
  mov( 0, ebx );
  while( ebx < eax ) do

    str.cat
    (
      "string[",
      (type uns32 ebx),
      "]=\"",
      strings[ebx*4],
      "\"",
      nl
    );
    strfree( strings[ebx*4] );
    inc( ebx );

  endwhile;

end tokenizeDemo;
```

This program produces the following output:

```
string[0]="This"
string[1]="string"
string[2]="contains"
string[3]="five"
string[4]="words"
```

This function raises an *ex.AttemptToDerefNULL* exception if *src* or *dest* contain NULL. It raises an *ex.AccessViolation* if *src* or *dest* contain an invalid address. This function raises an *ex.ArrayBounds* exception if it attempts to produce more than *maxStrs* lexemes while tokenizing the *src* string.

HLA high-level calling sequence examples:

```
static
  destArray:string[128];
  .
  .
  .
  str.tokenize3( someStr, destArray, 128 );
```

```

    mov( eax, numTokens );
    .
    .
    .
    for( mov( 0, ecx ); ecx < numTokens; inc( ecx )) do

        str.free( destArray[ ecx*4 ] );

    endfor;

```

HLA low-level calling sequence examples:

```

static
    destArray:string[128];
    .
    .
    .
push( someStr );
pushd( &destArray );
pushd( 128 );
    call str.tokenize3;
    mov( eax, numTokens );

```

```

procedure str.tokenize4
(
    src          :string;
    delimiters   :cset;
    var dest     :var;
    maxStrs     :dword
); @returns( "eax" );

```

This procedure is functionally identical to *str.tokenize3* except you get to specify the *delimiters* character set (rather than using the built-in *str.CmdLnDelimiters* character set) that the function uses to separate lexems in the *src* string. See the discussion of *str.tokenize3* for more details.

HLA high-level calling sequence examples:

```

static
    destArray:string[128];
    .
    .
    .
    str.tokenize4( someStr, chars.WhiteSpaceCset, destArray, 128 );
    mov( eax, numTokens );
    .
    .
    .
    for( mov( 0, ecx ); ecx < numTokens; inc( ecx )) do

        str.free( destArray[ ecx*4 ] );

    endfor;

```

HLA low-level calling sequence examples:

```

static
    destArray:string[128];
    .
    .
    .
push( someStr );
push( (type dword chars.WhiteSpaceCset[12]) );
push( (type dword chars.WhiteSpaceCset[ 8]) );
push( (type dword chars.WhiteSpaceCset[ 4]) );
push( (type dword chars.WhiteSpaceCset[ 0]) );
pushd( &destArray );
pushd( 128 );
    call str.tokenize4;
    mov( eax, numTokens );

```

**iterator str.tokenInStr( src:string );**

This iterator lexically scans the *src* string returns a pointer to a newly allocated lexeme on the heap (in EAX) on each foreach loop iteration. It is the caller's responsibility to free this storage when it is no longer needed. Like *str.tokenize3*, this iterator separates tokens in the input *src* string using the *str.CmdLnDelimiters* character set.

**Warning:** this function does not make a local copy of the *src* string to use during the execution of the invoking foreach loop. This iterator produces undefined results if the *src* string changes during the execution of the invoking foreach loop.

This function raises an *ex.AttemptToDerefNULL* exception if *src* contains NULL. It raises an *ex.AccessViolation* if *src* contains an invalid address.

HLA high-level calling sequence examples:

```

foreach str.tokenInStr( "This String Has 5 Words" ) do

    mov( eax, tokenStr );

    // Do something with the string pointed at by EAX/tokenStr
    .
    .
    .
    str.free( tokenStr );

endfor;

```

HLA low-level calling sequence examples:

(see the HLA reference manual for instructions on making low-level calls to iterators.)

**iterator str.tokenInStr2( src:string; delimiters:cset );**

This iterator lexically scans the *src* string returns a pointer to a newly allocated lexeme on the heap (in EAX) on each foreach loop iteration. It is the caller's responsibility to free this storage when it is no longer needed. Like *str.tokenize4*, this iterator separates tokens in the input *src* string using the *delimiters* character set passed as an argument.

**Warning:** this function does not make a local copy of the *src* string to use during the execution of the invoking foreach loop. This iterator produces undefined results if the *src* string changes during the execution of the invoking foreach loop.

This function raises an *ex.AttemptToDerefNULL* exception if *src* contains NULL. It raises an *ex.AccessViolation* if *src* contains an invalid address.

HLA high-level calling sequence examples:

```
foreach
    str.tokenInStr2
    (
        "This String Has 5 Words",
        chars.WhiteSpaceCset
    )
do

    mov( eax, tokenStr );

    // Do something with the string pointed at by EAX/tokenStr
    .
    .
    .
    str.free( tokenStr );

endfor;
```

HLA low-level calling sequence examples:

(see the HLA reference manual for instructions on making low-level calls to iterators.)

**iterator str.charInStr( src:string );**

This iterator scans the *src* string returns each successive character in the string in the AL register on each foreach loop iteration.

This function raises an *ex.AttemptToDerefNULL* exception if *src* contains NULL. It raises an *ex.AccessViolation* if *src* contains an invalid address.

HLA high-level calling sequence examples:

```
foreach str.charInStr( "abcdefghijklmnopqrstuvwxyz" ) do

    mov( al, currentChar );

    // Do something with the character in AL

endfor;
```

HLA low-level calling sequence examples:

(see the HLA reference manual for instructions on making low-level calls to iterators.)

**iterator str.wordInStr( src:string );**

This iterator lexically scans the *src* string returns a pointer to a locally (to the iterator) allocated word (in EAX) on each foreach loop iteration. The iterator will free this storage on the next iteration of the loop. If the caller needs to maintain the string value after the execution of the current loop iteration, the caller must allocate

storage for the string and make a copy of it. This iterator is similar to *str.tokenInStr* with two main differences: it separates tokens in the input *src* string using whitespace characters and this iterator makes a local copy of *src* before iterating to guarantee consistent results should *src* change during the execution of the invoking foreach loop.

This function raises an *ex.AttemptToDerefNULL* exception if *src* contains NULL. It raises an *ex.AccessViolation* if *src* contains an invalid address.

HLA high-level calling sequence examples:

```
foreach str.wordInStr( "This String Has 5 Words" ) do

    mov( eax, wordStr );

    // Do something with the string pointed at by EAX/wordStr
    .
    .
    .
    str.free( wordStr );

endfor;
```

HLA low-level calling sequence examples:

(see the HLA reference manual for instructions on making low-level calls to iterators.)

**procedure str.a\_getField2( src:string; field:dword ); @returns( "@c" );**

This function extracts a lexeme from the *src* string and returns a pointer to that lexeme (allocated on the heap) in EAX. The *field* parameter specifies which lexeme to extract from *src*. This function uses the *str.CmdLnDelimiters* ( { #0, ' ', #9, ',', '<', '>', '|', '\', '/', '-' } ) to specify the lexeme delimiter characters. This function returns with the carry flag set if it can locate and extract lexeme number *field*; it returns with the carry clear if there aren't *field* lexemes present in *src* (EAX's value is undefined in this case). It is the caller's responsibility to free up the storage allocated on the heap when the caller is done using the string data this function returns.

This function raises an *ex.AttemptToDerefNULL* exception if *src* contains NULL. It raises an *ex.AccessViolation* if *src* contains an invalid address. It raises an *ex.MemoryAllocation* exception if there is a problem allocating memory for the result string.

HLA high-level calling sequence examples:

```
if( str.a_getField2( someStr, 5 )) then

    mov( eax, lexeme );

    // Do something with the string pointed at by EAX/lexeme
    .
    .
    .
    str.free( lexeme );

endif;
```

HLA low-level calling sequence examples:

```
push( someStr );
```

```

pushd( 5 );
call str.a_getField2;
jnc noLexeme;

    mov( eax, lexeme );

    // Do something with the string pointed at by EAX/lexeme
    .
    .
    .
    str.free( lexeme );

noLexeme:

```

```

procedure str.a_getField3( src:string; field:dword; delimiters:cset );
  @returns( "@c" );

```

This function works just like *str.a\_getField2* with the additional capability of being able to specify the lexeme delimiter character set (in the *delimiters* parameter).

This function raises an *ex.AttemptToDerefNULL* exception if *src* contains NULL. It raises an *ex.AccessViolation* if *src* contains an invalid address. It raises an *ex.MemoryAllocation* exception if there is a problem allocating memory for the result string.

HLA high-level calling sequence examples:

```

if( str.a_getField3( someStr, 5, chars.WhiteSpaceCset ) ) then

    mov( eax, lexeme );

    // Do something with the string pointed at by EAX/lexeme
    .
    .
    .
    str.free( lexeme );

endif;

```

HLA low-level calling sequence examples:

```

push( someStr );
pushd( 5 );
push( (type dword chars.WhiteSpaceCset[12]) );
push( (type dword chars.WhiteSpaceCset[ 8]) );
push( (type dword chars.WhiteSpaceCset[ 4]) );
push( (type dword chars.WhiteSpaceCset[ 0]) );
call str.a_getField3;
jnc noLexeme;

    mov( eax, lexeme );

    // Do something with the string pointed at by EAX/lexeme
    .
    .
    .
    str.free( lexeme );

noLexeme:

```



```

procedure str.getField3( src:string; field:dword; dest:string );
    @returns( "@c" );

```

This function extracts a lexeme from the *src* string and stores that string in the object pointed at by *dest*. The *field* parameter specifies which lexeme to extract from *src*. This function uses the *str.CmdLnDelimiters* ( { #0, ' ', #9, ',', '<', '>', '|', '\', '/', '-' } ) to specify the lexeme delimiter characters. This function returns with the carry flag set if it can locate and extract lexeme number *field*; it returns with the carry clear if there aren't *field* lexemes present in *src*.

This function raises an *ex.AttemptToDerefNULL* exception if *src* or *dest* contain NULL. It raises an *ex.AccessViolation* if *src* or *dest* contain an invalid address. It raises an *ex.StringOverflow* exception if the string object pointed at by *dest* isn't large enough to hold the extracted lexeme.

HLA high-level calling sequence examples:

```

    if( str.getField3( someStr, 5, lexeme ) ) then
        // Do something with the string pointed at by lexeme
    endif;

```

HLA low-level calling sequence examples:

```

    push( someStr );
    pushd( 5 );
    push( lexeme );
    call str.getField3;
    jnc noLexeme;

    // Do something with the string pointed at by lexeme

noLexeme:

```

```

procedure str.getField4
(
    src           :string;
    field         :dword;
    delimiters    :cset;
    dest          :string
); @returns( "@c" );

```

This function extracts a lexeme from the *src* string and stores that string in the object pointed at by *dest*. The *field* parameter specifies which lexeme to extract from *src*. This function uses the *delimiters* character set to specify the lexeme delimiter characters. This function returns with the carry flag set if it can locate and extract lexeme number *field*; it returns with the carry clear if there aren't *field* lexemes present in *src*.

This function raises an *ex.AttemptToDerefNULL* exception if *src* or *dest* contain NULL. It raises an *ex.AccessViolation* if *src* or *dest* contain an invalid address. It raises an *ex.StringOverflow* exception if the string object pointed at by *dest* isn't large enough to hold the extracted lexeme.

HLA high-level calling sequence examples:

```

    if( str.getField4( someStr, 5, chars.WhiteSpaceCset, lexeme ) ) then

```

```

        // Do something with the string pointed at by lexeme

    endif;

```

HLA low-level calling sequence examples:

```

push( someStr );
pushd( 5 );
push( (type dword chars.WhiteSpaceCset[12]) );
push( (type dword chars.WhiteSpaceCset[ 8]) );
push( (type dword chars.WhiteSpaceCset[ 4]) );
push( (type dword chars.WhiteSpaceCset[ 0]) );
push( lexeme );
call str.getField4;
jnc noLexeme;

```

```

        // Do something with the string pointed at by lexeme

noLexeme:

```

**procedure str.rmv1stChar1( s:string );@returns( "al" );**

This function deletes the first character (in-place) from the *s* string and returns that character in AL. Note that on return this first character is no longer present in the *s* string. On entry, if *s* is the empty string, this function returns zero in AL and does not otherwise affect the *s* string.

This function raises an *ex.AttemptToDerefNULL* exception if *src* contains NULL. It raises an *ex.AccessViolation* if *src* contains an invalid address.

HLA high-level calling sequence examples:

```

if( str.rmv1stChar1( someStr ) <> #0) then

    // Do something with the char in AL

endif;

```

HLA low-level calling sequence examples:

```

push( someStr );
call str.rmv1stChar1;
cmp( al, 0 );
je noChar;

        // Do something with the char in AL

noChar:

```

```
procedure str.rmv1stChar2( src:string; remainder:string );@returns( "al" );
```

This function returns the first character (if any) of *src* in the AL register and copies any remaining characters in *src* to the *remainder* string object. Note that if *src* is the empty string, this function returns zero in AL and does not otherwise affect the *remainder* string.

This function raises an *ex.AttemptToDerefNULL* exception if *src* or *remainder* contain NULL. It raises an *ex.AccessViolation* if *src* or *remainder* contain an invalid address. It raises an *ex.StringOverflow* exception if the string object pointed at by *remainder* isn't large enough to hold the result.

HLA high-level calling sequence examples:

```
if( str.rmv1stChar2( someStr, dest ) <> #0 ) then

    // Do something with the char in AL and the
    // string in dest.

endif;
```

HLA low-level calling sequence examples:

```
push( someStr );
push( dest );
call str.rmv1stChar2;
cmp( al, 0 );
je noChar;

    // Do something with the char in AL

noChar:
```

```
procedure str.rmvLastChar1( s:string );@returns( "al" );
```

This function deletes the last character (in-place) from the *s* string and returns that character in AL. Note that on return this last character is no longer present in the *s* string. On entry, if *s* is the empty string, this function returns zero in AL and does not otherwise affect the *s* string.

This function raises an *ex.AttemptToDerefNULL* exception if *src* contains NULL. It raises an *ex.AccessViolation* if *src* contains an invalid address.

HLA high-level calling sequence examples:

```
if( str.rmvLastChar1( someStr ) <> #0 ) then

    // Do something with the char in AL

endif;
```

HLA low-level calling sequence examples:

```
push( someStr );
call str.rmvLastChar1;
cmp( al, 0 );
je noChar;

    // Do something with the char in AL
```

noChar:

```
procedure str.rmvLastChar2( src:string; remainder:string );@returns( "al" );
```

This function returns the last character (if any) of *src* in the AL register and copies any previous characters in *src* to the *remainder* string object. Note that if *src* is the empty string, this function returns zero in AL and does not otherwise affect the *remainder* string.

This function raises an *ex.AttemptToDerefNULL* exception if *src* or *remainder* contain NULL. It raises an *ex.AccessViolation* if *src* or *remainder* contain an invalid address. It raises an *ex.StringOverflow* exception if the string object pointed at by *remainder* isn't large enough to hold the result.

HLA high-level calling sequence examples:

```
if( str.rmvLastChar2( someStr, dest ) <> #0 ) then

    // Do something with the char in AL and the
    // string in dest.

endif;
```

HLA low-level calling sequence examples:

```
push( someStr );
push( dest );
call str.rmvLastChar2;
cmp( al, 0 );
je noChar;

    // Do something with the char in AL and the
    // string in dest.

noChar:
```

```
procedure str.a_rmv1stWord1( s:string );@returns( "@c" );
```

This function deletes the first word (in-place) from the *s* string, copies that word to the string object allocated on the heap (pointer returned in EAX), and returns with the carry flag set. It is the caller's responsibility to free the storage when it is no longer needed. Note that on return this first word is no longer present in the *s* string. On entry, if *s* is the empty string, this function returns with the carry clear and does not otherwise affect the *s* string (EAX is undefined in this case).

This function raises an *ex.AttemptToDerefNULL* exception if *src* contains NULL. It raises an *ex.AccessViolation* if *src* contains an invalid address. It raises an *ex.MemoryAllocation* exception if there is a problem allocating memory for the result string.

HLA high-level calling sequence examples:

```
if( str.a_rmv1stWord1( someStr ) ) then

    // Do something with the string pointed at by EAX.

endif;
```

HLA low-level calling sequence examples:

```

push( someStr );
call str.a_rmvlstWord1;
jnc noWord;

    // Do something with the string pointed at by EAX.

noWord:

```

```

procedure str.a_rmvlstWord2( src:string; remainder:string );
    @returns( "@c" );

```

This function copies the first word (if any) of *src* to storage it allocates on the heap (pointer returned in EAX), copies any remaining characters in *src* to the *remainder* string object, and sets the carry flag. Note that if *src* is the empty string, this function returns with the carry flag clear, EAX undefined, and does not affect *remainder* string.

This function raises an *ex.AttemptToDerefNULL* exception if *src*, or *remainder* contain NULL. It raises an *ex.AccessViolation* if *src*, or *remainder* contain an invalid address. It raises an *ex.StringOverflow* exception if the string object pointed at by *remainder* isn't large enough to hold the result. It raises an *ex.MemoryAllocation* exception if there is a problem allocating memory for the result string.

HLA high-level calling sequence examples:

```

    if( str.a_rmvlstWord2( someStr, remainder ) ) then

        mov( eax, wordStr );

        // Do something with the strings in EAX and remainder.
        .
        .
        .
        str.free( wordStr );

    endif;

```

HLA low-level calling sequence examples:

```

push( someStr );
push( remainder );
call str.a_rmvlstWord2;
jnc noWord;

    mov( eax, wordStr );

    // Do something with the strings in EAX and remainder.
    .
    .
    .
    str.free( wordStr );

noWord:

```

```
procedure str.a_rmvlstWord1( s:string );@returns( "@c" );
```

This function deletes the last word (in-place) from the *s* string, copies that word to the string object allocated on the heap (pointer returned in EAX), and returns with the carry flag set. It is the caller's responsibility to free the storage when it is no longer needed. Note that on return this first word is no longer present in the *s* string. On entry, if *s* is the empty string, this function returns with the carry clear and does not otherwise affect the *s* string (EAX is undefined in this case).

This function raises an *ex.AttemptToDerefNULL* exception if *src* contains NULL. It raises an *ex.AccessViolation* if *src* contains an invalid address. It raises an *ex.MemoryAllocation* exception if there is a problem allocating memory for the result string.

HLA high-level calling sequence examples:

```
if( str.a_rmvlstWord1( someStr, wordResult ) <> #0 ) then

    mov( eax, wordStr );

    // Do something with the string pointed at by EAX.
    .
    .
    .
    str.free( wordStr );

endif;
```

HLA low-level calling sequence examples:

```
push( someStr );
call str.a_rmvlstWord1;
jnc noWord;

    mov( eax, wordStr );

    // Do something with the string in EAX.
    .
    .
    .
    str.free( wordStr );

noWord:
```

```
procedure str.a_rmvlstWord2( src:string; remainder:string );
    @returns( "@c" );
```

This function returns the last word (if any) of *src* in storage allocated on the heap (pointer returned in EAX), copies any remaining characters in *src* to the *remainder* string object, and sets the carry flag. Note that if *src* is the empty string, this function returns with the carry flag clear, EAX undefined, and does not affect the *remainder* string.

This function raises an *ex.AttemptToDerefNULL* exception if *src* or *remainder* contain NULL. It raises an *ex.AccessViolation* if *src* or *remainder* contain an invalid address. It raises an *ex.StringOverflow* exception if the string object pointed at by *remainder* isn't large enough to hold the result. It raises an *ex.MemoryAllocation* exception if there is a problem allocating memory for the result string.

HLA high-level calling sequence examples:

```
if( str.a_rmvlstWord2( someStr, dest ) ) then
```

```

    mov( eax, wordStr );

    // Do something with the strings in EAX and dest.
    .
    .
    .
    str.free( wordStr );

endif;

```

HLA low-level calling sequence examples:

```

push( someStr );
push( dest );
call str.a_rmvLastWord2;
jnc noWord;

    mov( eax, wordStr );

    // Do something with the strings in EAX and dest.
    .
    .
    .
    str.free( wordStr );

noWord:

```

**procedure str.rmv1stWord2( s:string; wordStr:string );@returns( "@c" );**

This function deletes the first word (in-place) from the *s* string, copies that word to the string object pointed at by *wordStr*, and returns with the carry flag set. Note that on return this first word is no longer present in the *s* string. On entry, if *s* is the empty string, this function returns with the carry clear and does not otherwise affect the *s* string (*wordStr* is not modified in this case).

This function raises an *ex.AttemptToDerefNULL* exception if *src* or *wordStr* contain NULL. It raises an *ex.AccessViolation* if *src* or *wordStr* contain an invalid address.

HLA high-level calling sequence examples:

```

if( str.rmv1stWord2( someStr, wordResult ) ) then

    // Do something with the wordResult string

endif;

```

HLA low-level calling sequence examples:

```

push( someStr );
push( wordResult );
call str.rmv1stWord2;
jnc noWord;

    // Do something with the wordResult string

```

noWord:

```
procedure str.rmv1stWord3( src:string; wordStr:string; remainder:string );
  @returns( "@c" );
```

This function returns the first word (if any) of *src* in *wordStr*, copies any remaining characters in *src* to the *remainder* string object, and sets the carry flag. Note that if *src* is the empty string, this function returns with the carry flag clear and does not affect the *wordStr* or *remainder* strings.

This function raises an *ex.AttemptToDerefNULL* exception if *src*, *wordStr*, or *remainder* contain NULL. It raises an *ex.AccessViolation* if *src*, *wordStr*, or *remainder* contain an invalid address. It raises an *ex.StringOverflow* exception if the string objects pointed at by *wordStr* or *remainder* aren't large enough to hold their respective results.

HLA high-level calling sequence examples:

```
if( str.rmv1stWord3( someStr, dest, remainder ) <> #0 ) then
    // Do something with the strings in dest and remainder.
endif;
```

HLA low-level calling sequence examples:

```
push( someStr );
push( dest );
push( remainder );
call str.rmv1stWord3;
cmp( al, 0 );
je noWord;

// Do something with the strings in dest and remainder.

noWord:
```

```
procedure str.rmvLastWord2( s:string; wordStr:string );@returns( "@c" );
```

This function deletes the last word (in-place) from the *s* string, copies that word to the string object pointed at by *wordStr*, and returns with the carry flag set. Note that on return this first word is no longer present in the *s* string. On entry, if *s* is the empty string, this function returns with the carry clear and does not otherwise affect the *s* string (*wordStr* is not modified in this case).

This function raises an *ex.AttemptToDerefNULL* exception if *src* or *wordStr* contain NULL. It raises an *ex.AccessViolation* if *src* or *wordStr* contain an invalid address.

HLA high-level calling sequence examples:

```
if( str.rmvLastWord2( someStr, wordResult ) <> #0 ) then
    // Do something with the strings in someStr and wordResult.
endif;
```

HLA low-level calling sequence examples:



```

push( someStr );
push( wordResult );
call str.rmvLastWord2;
cmp( al, 0 );
je noWord;

    // Do something with the strings in someStr and wordResult.

noWord:

```

```

procedure str.rmvLastWord3( src:string; wordStr:string; remainder:string );
  @returns( "@c" );

```

This function returns the last word (if any) of *src* in *wordStr*, copies any remaining characters in *src* to the *remainder* string object, and sets the carry flag. Note that if *src* is the empty string, this function returns with the carry flag clear and does not affect the *wordStr* or *remainder* strings.

This function raises an *ex.AttemptToDerefNULL* exception if *src*, *wordStr*, or *remainder* contain NULL. It raises an *ex.AccessViolation* if *src*, *wordStr*, or *remainder* contain an invalid address. It raises an *ex.StringOverflow* exception if the string objects pointed at by *wordStr* or *remainder* aren't large enough to hold their respective results.

HLA high-level calling sequence examples:

```

if( str.rmvLastWord3( someStr, wordResult, dest )) then

    // Do something with the strings in dest and wordResult.

endif;

```

HLA low-level calling sequence examples:

```

push( someStr );
push( wordResult );
push( dest );
call str.rmvLastWord3;
cmp( al, 0 );
je noWord;

    // Do something with the strings in dest and wordResult.

noWord:

```

## 33.11 String Formatting Functions

The HLA Standard Library provides a couple of routines that can be used to format the data appearing in a string.

```

procedure str.a_columnize2( var s:var; numStrs:dword );
    @returns( "eax" );

```

This function scans an array of *numStrs* string pointed at by *s* and computes the maximum length of all the strings. This function then creates a single string on the heap that consists of the concatenation of all the strings in *s* with their lengths extended to the maximum string length in *s* plus one. The extra character positions at the end of each string are padded with spaces.

This function raises an *ex.AttemptToDerefNULL* exception if *s* contains NULL. It raises an *ex.AccessViolation* if *s* contains an invalid address. It raises an *ex.MemoryAllocation* exception if there is an error allocating storage for the resulting string.

HLA high-level calling sequence examples:

```

    str.a_columnize2( stringArray, 10 );
    mov( eax, columnsStr );

    // Do something with the string pointed at by EAX
    .
    .
    .
    str.free( columnsStr );

```

HLA low-level calling sequence examples:

```

    push( stringArray );
    pushd( 10 );
    call str.a_columnize2;
    mov( eax, columnsStr );

    // Do something with the string pointed at by EAX
    .
    .
    .
    str.free( columnsStr );

```

```

procedure str.a_columnize3( var s:var; numStrs:dword; tabCols:dword );
    @returns( "eax" );

```

This function scans an array of *numStrs* string pointed at by *s* and creates a single string on the heap that consists of the concatenation of all the strings in *s* with their lengths extended to *tabCols*. The extra character positions at the end of each string are padded with spaces.

This function raises an *ex.AttemptToDerefNULL* exception if *s* contains NULL. It raises an *ex.AccessViolation* if *s* contains an invalid address. It raises an *ex.MemoryAllocation* exception if there is an error allocating storage for the resulting string.

HLA high-level calling sequence examples:

```

    str.a_columnize3( stringArray, 10, 40 );
    mov( eax, columnsStr );

    // Do something with the string pointed at by EAX
    .
    .
    .
    str.free( columnsStr );

```

HLA low-level calling sequence examples:

```

push( stringArray );
pushd( 10 );
pushd( 40 );
call str.a_columnize3;
    mov( eax, columnsStr );

    // Do something with the string pointed at by EAX
    .
    .
    .
str.free( columnsStr );

```

**procedure str.columnize3( var s:var; numStrs:dword; dest:string );**

This function scans an array of *numStrs* string pointed at by *s* and computes the maximum length of all the strings. This function then creates a single string that it stores in the string object pointed at by *dest* which consists of the concatenation of all the strings in *s* with their lengths extended to the maximum string length in *s* plus one. The extra character positions at the end of each string are padded with spaces.

This function raises an *ex.AttemptToDerefNULL* exception if *s* contains NULL. It raises an *ex.AccessViolation* if *s* contains an invalid address.

HLA high-level calling sequence examples:

```

str.columnize3( stringArray, 10, columnsStr );

// Do something with the string pointed at by columnsStr

```

HLA low-level calling sequence examples:

```

push( stringArray );
pushd( 10 );
push( columnsStr );
call str.a_columnize3;

// Do something with the string pointed at by columnsStr

```

```

procedure str.columnize4
(
    var    s           :var;
           numStrs      :dword;
           tabCols     :dword;
           dest        :string
);

```

This function scans an array of *numStrs* string pointed at by *s* and creates a single string it stores in *dest* that consists of the concatenation of all the strings in *s* with their lengths extended to *tabCols*. The extra character positions at the end of each string are padded with spaces.

This function raises an *ex.AttemptToDerefNULL* exception if *s* contains NULL. It raises an *ex.AccessViolation* if *s* contains an invalid address. It raises an *ex.MemoryAllocation* exception if there is an error allocating storage for the resulting string.

HLA high-level calling sequence examples:

```
str.columnize4( stringArray, 10, 40, columnsStr );

// Do something with the string pointed at by columnsStr
```

HLA low-level calling sequence examples:

```
push( stringArray );
pushd( 10 );
pushd( 40 );
push( columnsStr );
call str.columnize4;

// Do something with the string pointed at by columnsStr
```

```
procedure str.a_spread2( src:string; toWidth:dword );
@returns( "@c" );
```

This creates a new string on the heap (returning the pointer in EAX) that is an expansion of the *src* string to the length specified by the *toWidth* parameter. If the length of *src* is greater than or equal to *toWidth*, then this function clears the carry flag and returns with EAX containing NULL; otherwise, this function fills in the extra character positions using space characters and returns with the carry flag set. If the length of *src* is greater than 75% of *toWidth*, then this function pads the end of the result string with spaces to fill in the extra length. If the length of *src* is 75% or less of *toWidth*, then this function spreads the space characters throughout the string (next to other spaces appearing in the *src* string) to widen the resulting string. It is the caller's responsibility to free the storage allocated on the heap if this function returns with the carry flag set.

This function raises an *ex.AttemptToDerefNULL* exception if *src* contains NULL. It raises an *ex.AccessViolation* if *src* contains an invalid address. It raises an *ex.MemoryAllocation* exception if there is a problem allocating memory for the result string.

HLA high-level calling sequence examples:

```
if( str.a_spread2( someStr, newLength ) ) then

    mov( eax, newStr );

    // Do something with newStr
    .
    .
    .
    str.free( newStr );

endif;
```

HLA low-level calling sequence examples:

```
push( someStr );
push( newLength );
call str.a_spread2;
jnc noNewStr;

mov( eax, newStr );

// Do something with newStr
.
```

```

        .
        .
    str.free( newStr );

noNewStr:

```

```

procedure str.spread2( s:string; toWidth:dword );
    @returns( "@c" );

```

This expands the *s* string to the length specified by the *toWidth* parameter. If the length of *s* is greater than or equal to *toWidth*, then this function clears the carry flag and does not modify *s*; otherwise, this function fills in the extra character positions using space characters and returns with the carry flag set. If the length of *s* is greater than 75% of *toWidth*, then this function pads the end of *s* with spaces to fill in the extra length. If the length of *s* is 75% or less of *toWidth*, then this function spreads the space characters throughout the string (next to other spaces appearing in the *s* string) to widen the resulting string.

This function raises an *ex.AttemptToDerefNULL* exception if *s* contains NULL. It raises an *ex.AccessViolation* if *s* contains an invalid address.

HLA high-level calling sequence examples:

```

    if( str.spread2( someStr, newLength )) then

        // Do something with expanded someStr

    endif;

```

HLA low-level calling sequence examples:

```

push( someStr );
push( newLength );
call str.spread2;
jnc noExpandedStr;

    // Do something with expanded someStr

noExpandedStr:

```

```

procedure str.spread3( src:string; toWidth:dword; dest:string );
    @returns( "@c" );

```

This expands the *src* string to the length specified by the *toWidth* parameter and stores the result in the string object pointed at by *dest*. If the length of *src* is greater than or equal to *toWidth*, then this function clears the carry flag and does not modify *dest*; otherwise, this function fills in the extra character positions using space characters and returns with the carry flag set. If the length of *src* is greater than 75% of *toWidth*, then this function pads the end of *dest* with spaces to fill in the extra length. If the length of *src* is 75% or less of *toWidth*, then this function spreads the space characters throughout the *dest* string (next to other spaces appearing in the string) to widen the resulting string.

This function raises an *ex.AttemptToDerefNULL* exception if *src* or *dest* contain NULL. It raises an *ex.AccessViolation* if *src* or *dest* contain an invalid address. It raises an *ex.StringOverflow* exception if the string object pointed at by *dest* is not large enough to receive the result.

HLA high-level calling sequence examples:

```

    if( str.spread3( someStr, length, expandedStr )) then

        // do something with expandedStr
    endif;

```

```
endif;
```

HLA low-level calling sequence examples:

```
push( someStr );
push( length );
push( expandedStr );
call str.spread3;
jnc noExpandedStr;

    // do something with expandedStr

noExpandedStr:
```

```
procedure str.a_deTab2( src:string; tabCols:dword );
    @returns( "(type string eax)" );
```

This function creates a new string on the heap (returning the pointer in EAX) that is an expansion of the *src* string by converting all tab characters to the corresponding number of spaces. The *tabCols* argument specifies the number of character positions for each tab stop (all tab stops are equal in length). It is the caller's responsibility to free the storage allocated on the heap.

This function raises an *ex.AttemptToDerefNULL* exception if *src* contains NULL. It raises an *ex.AccessViolation* if *src* contains an invalid address. It raises an *ex.MemoryAllocation* exception if there is a problem allocating memory for the result string.

HLA high-level calling sequence examples:

```
str.a_deTab2( someStr, 4 );
mov( eax, newStr );

// Do something with newStr
.
.
.
str.free( newStr );
```

HLA low-level calling sequence examples:

```
push( someStr );
pushd( 4 );
call str.a_deTab2;
    mov( eax, newStr );

// Do something with newStr
.
.
.
str.free( newStr );
```

```

procedure str.a_deTab3( src:string; var tabCols:var; numTabs:dword );
    @returns( "(type string eax)" );

```

This function creates a new string on the heap (returning the pointer in EAX) that is an expansion of the *src* string by converting all tab characters to the corresponding number of spaces. The *tabCols* argument is an array of tabstop column values to use. The *numTabs* parameter specifies the total number of tabstops present in the *tabCols* array. It is the caller's responsibility to free the storage allocated on the heap.

This function raises an *ex.AttemptToDerefNULL* exception if *src* or *tabCols* contain NULL. It raises an *ex.AccessViolation* if *src* or *tabCols* contain an invalid address. It raises an *ex.MemoryAllocation* exception if there is a problem allocating memory for the result string.

HLA high-level calling sequence examples:

```

static
    tabStops:dword[4] := [4, 12, 16, 32];
    .
    .
    .
    str.a_deTab3( someStr, tabStops, 4 );
    mov( eax, newStr );

    // Do something with newStr
    .
    .
    .
    str.free( newStr );

```

HLA low-level calling sequence examples:

```

static
    tabStops:dword[4] := [4, 12, 16, 32];
    .
    .
    .
    push( someStr );
    pushd( &tabStops );
    pushd( 4 );
    call str.a_deTab3;
    mov( eax, newStr );

    // Do something with newStr
    .
    .
    .
    str.free( newStr );

```

```

procedure str.deTab2( s:string; tabCols:dword );

```

This function expands the *s* string by converting all tab characters to the corresponding number of spaces. The *tabCols* argument specifies the number of character positions for each tab stop (all tab stops are equal in length).

This function raises an *ex.AttemptToDerefNULL* exception if *s* contains NULL. It raises an *ex.AccessViolation* if *s* contains an invalid address. It raises an *ex.StringOverflow* exception the string object pointed at by *s* is not large enough to hold the result.

HLA high-level calling sequence examples:

```
str.deTab2( someStr, 4 );

// Do something with someStr
```

HLA low-level calling sequence examples:

```
push( someStr );
pushd( 4 );
call str.deTab2;

// Do something with someStr
```

**procedure str.deTab3a( src:string; tabCols:dword; dest:string );**

This function expands the *src* string by converting all tab characters to the corresponding number of spaces, it stores the result into the string object pointed at by *dest*. The *tabCols* argument specifies the number of character positions for each tab stop (all tab stops are equal in length).

This function raises an *ex.AttemptToDerefNULL* exception if *src* or *dest* contain NULL. It raises an *ex.AccessViolation* if *src* or *dest* contain an invalid address. It raises an *ex.StringOverflow* exception if the string object pointed at by *dest* is not large enough to hold the expanded result.

HLA high-level calling sequence examples:

```
str.deTab3a( someStr, 8, destStr );

// Do something with destStr
```

HLA low-level calling sequence examples:

```
push( someStr );
pushd( 8 );
push( destStr );
call str.deTab3a;

// Do something with destStr
```

**procedure str.deTab3b( s:string; var tabCols:var; numTabs:dword );**

This function expands the *s* string by converting all tab characters to the corresponding number of spaces. The *tabCols* argument is an array of tabstop column values to use. The *numTabs* parameter specifies the total number of tabstops present in the *tabCols* array.

This function raises an *ex.AttemptToDerefNULL* exception if *s* or *tabCols* contain NULL. It raises an *ex.AccessViolation* if *s* or *tabCols* contain an invalid address. It raises an *ex.StringOverflow* exception if the string object pointed at by *s* is not large enough to hold the expanded result.

HLA high-level calling sequence examples:

```
static
  tabStops:dword[4] := [4, 12, 16, 32];
  .
  .
  .
  str.deTab3b( someStr, tabStops, 4 );
```



```
// Do something with someStr
```

HLA low-level calling sequence examples:

```
static
  tabStops:dword[4] := [4, 12, 16, 32];
  .
  .
  .
push( someStr );
pushd( &tabStops );
pushd( 4 );
call str.deTab3b;

// Do something with someStr
```

```
procedure str.deTab4
(
    src          :string;
var   tabCols   :var;
       numTabs    :dword;
       dest       :string
);
```

This function expands the *src* string by converting all tab characters to the corresponding number of spaces, it stores the result into the string object pointed at by *dest*. The *tabCols* argument is an array of tabstop column values to use. The *numTabs* parameter specifies the total number of tabstops present in the *tabCols* array.

This function raises an *ex.AttemptToDerefNULL* exception if *src*, *dest*, or *tabCols* contain NULL. It raises an *ex.AccessViolation* if *src*, *dest*, or *tabCols* contain an invalid address. It raises an *ex.StringOverflow* exception if the string object pointed at by *dest* is not large enough to hold the expanded result.

HLA high-level calling sequence examples:

```
static
  tabStops:dword[4] := [4, 12, 16, 32];
  .
  .
  .
str.deTab4( someStr, tabStops, 4, destStr );

// Do something with destStr
```

HLA low-level calling sequence examples:

```
static
  tabStops:dword[4] := [4, 12, 16, 32];
  .
  .
  .
push( someStr );
pushd( &tabStops );
pushd( 4 );
```

```

push( destStr );
call str.deTab4;

// Do something with destStr

```

```

procedure str.a_enTab2( src:string; tabCols:dword );
  @returns( "(type string eax)" );

```

This function creates a new string on the heap (returning the pointer in EAX) that is an expansion of the *src* string by converting all space characters to the corresponding number of tabs. The *tabCols* argument specifies the number of character positions for each tab stop (all tab stops are equal in length). It is the caller's responsibility to free the storage allocated on the heap.

This function raises an *ex.AttemptToDerefNULL* exception if *src* contains NULL. It raises an *ex.AccessViolation* if *src* contains an invalid address. It raises an *ex.MemoryAllocation* exception if there is a problem allocating memory for the result string.

HLA high-level calling sequence examples:

```

str.a_enTab2( someStr, 4 );
mov( eax, newStr );

// Do something with newStr
.
.
.
str.free( newStr );

```

HLA low-level calling sequence examples:

```

push( someStr );
pushd( 4 );
call str.a_enTab2;
mov( eax, newStr );

// Do something with newStr
.
.
.
str.free( newStr );

```

```

procedure str.a_enTab3( src:string; var tabCols:var; numTabs:dword );
  @returns( "(type string eax)" );

```

This function creates a new string on the heap (returning the pointer in EAX) that is an expansion of the *src* string by converting all space characters to the corresponding number of tabs. The *tabCols* argument is an array of tabstop column values to use. The *numTabs* parameter specifies the total number of tabstops present in the *tabCols* array. It is the caller's responsibility to free the storage allocated on the heap.

This function raises an *ex.AttemptToDerefNULL* exception if *src* or *tabCols* contain NULL. It raises an *ex.AccessViolation* if *src* or *tabCols* contain an invalid address. It raises an *ex.MemoryAllocation* exception if there is a problem allocating memory for the result string.

HLA high-level calling sequence examples:

```

static

```

```

tabStops:dword[4] := [4, 12, 16, 32];
.
.
.
str.a_enTab3( someStr, tabStops, 4 );
mov( eax, newStr );

// Do something with newStr
.
.
.
str.free( newStr );

```

HLA low-level calling sequence examples:

```

static
  tabStops:dword[4] := [4, 12, 16, 32];
.
.
.
push( someStr );
pushd( &tabStops );
pushd( 4 );
call str.a_enTab3;
  mov( eax, newStr );

// Do something with newStr
.
.
.
str.free( newStr );

```

#### **procedure str.enTab2( s:string; tabCols:dword );**

This function expands the *s* string by converting all space characters to the corresponding number of tabs. The *tabCols* argument specifies the number of character positions for each tab stop (all tab stops are equal in length).

This function raises an *ex.AttemptToDerefNULL* exception if *s* contains NULL. It raises an *ex.AccessViolation* if *s* contains an invalid address.

HLA high-level calling sequence examples:

```

str.enTab2( someStr, 4 );

// Do something with someStr

```

HLA low-level calling sequence examples:

```

push( someStr );
pushd( 4 );
call str.enTab2;

// Do something with someStr

```

```
procedure str.enTab3a( src:string; tabCols:dword; dest:string );
```

This function expands the *src* string by converting all space characters to the corresponding number of tabs, it stores the result into the string object pointed at by *dest*. The *tabCols* argument specifies the number of character positions for each tab stop (all tab stops are equal in length).

This function raises an *ex.AttemptToDerefNULL* exception if *src* or *dest* contain NULL. It raises an *ex.AccessViolation* if *src* or *dest* contain an invalid address. It raises an *ex.StringOverflow* exception if the string object pointed at by *dest* is not large enough to hold the expanded result.

HLA high-level calling sequence examples:

```
str.enTab3a( someStr, 8, destStr );

// Do something with destStr
```

HLA low-level calling sequence examples:

```
push( someStr );
pushd( 8 );
push( destStr );
call str.enTab3a;

// Do something with destStr
```

```
procedure str.enTab3b( s:string; var tabCols:var; numTabs:dword );
```

This function expands the *s* string by converting all space characters to the corresponding number of tabs. The *tabCols* argument is an array of tabstop column values to use. The *numTabs* parameter specifies the total number of tabstops present in the *tabCols* array.

This function raises an *ex.AttemptToDerefNULL* exception if *s* or *tabCols* contain NULL. It raises an *ex.AccessViolation* if *s* or *tabCols* contain an invalid address.

HLA high-level calling sequence examples:

```
static
  tabStops:dword[4] := [4, 12, 16, 32];
  .
  .
  .
str.enTab3b( someStr, tabStops, 4 );

// Do something with someStr
```

HLA low-level calling sequence examples:

```
static
  tabStops:dword[4] := [4, 12, 16, 32];
  .
  .
  .
push( someStr );
pushd( &tabStops );
pushd( 4 );
call str.enTab3b;
```

```
// Do something with someStr
```

```
procedure str.enTab4
(
    src          :string;
    var tabCols   :var;
    numTabs       :dword;
    dest          :string
);
```

This function expands the *src* string by converting all space characters to the corresponding number of tabs, it stores the result into the string object pointed at by *dest*. The *tabCols* argument is an array of tabstop column values to use. The *numTabs* parameter specifies the total number of tabstops present in the *tabCols* array.

This function raises an *ex.AttemptToDerefNULL* exception if *src*, *dest*, or *tabCols* contain NULL. It raises an *ex.AccessViolation* if *src*, *dest*, or *tabCols* contain an invalid address. It raises an *ex.StringOverflow* exception if the string object pointed at by *dest* is not large enough to hold the expanded result.

HLA high-level calling sequence examples:

```
static
    tabStops:dword[4] := [4, 12, 16, 32];
    .
    .
    .
    str.enTab4( someStr, tabStops, 4, destStr );

// Do something with destStr
```

HLA low-level calling sequence examples:

```
static
    tabStops:dword[4] := [4, 12, 16, 32];
    .
    .
    .
push( someStr );
pushd( &tabStops );
pushd( 4 );
push( destStr );
call str.enTab4;

// Do something with destStr
```

## 33.12 String Conversion Functions

The functions in this category transform string data from one form to another (e.g., upper case conversion).

```
procedure str.a_upper( src:string; dest:string );
```

This function scans the *src* string and converts all lower-case alphabetic characters to their uppercase equivalent. It stores the result into a new string it allocates on the heap (and returns a pointer to this string in EAX). It is the caller's responsibility to free this storage when it is no longer needed.

This function raises an *ex.AttemptToDerefNULL* exception if *src* contains NULL. It raises an *ex.AccessViolation* if *src* contains an invalid address. It raises an *ex.MemoryAllocation* exception if there is an error allocating storage to hold the result.

HLA high-level calling sequence examples:

```
str.a_upper( someStr );
mov( eax, upperStr );

// Do something with upperStr
.
.
.
str.free( upperStr );
```

HLA low-level calling sequence examples:

```
push( someStr );
call str.a_upper;
mov( eax, upperStr );

// Do something with upperStr
.
.
.
str.free( upperStr );
```

```
#macro upper( string );
```

```
#macro upper( string, string );
```

This macro provides a "function overload" declaration for the *str.upper1* and *str.upper2* functions. If you pass this macro one argument, it creates a call to the *str.upper1* function; if you pass this macro two arguments, it calls the *str.upper2* function.

```
procedure str.upper1( s:string );
```

This function scans the *s* string and converts, in-place, all lower-case alphabetic characters to their uppercase equivalent.

This function raises an *ex.AttemptToDerefNULL* exception if *s* contains NULL. It raises an *ex.AccessViolation* if *s* contains an invalid address.

HLA high-level calling sequence examples:

```
str.upper1( someStr );

// Do something with someStr
```

HLA low-level calling sequence examples:

```
push( someStr );
call str.upper1;

// Do something with someStr
```

```
procedure str.upper2( src:string; dest:string );
```

This function scans the *src* string and converts all lower-case alphabetic characters to their uppercase equivalent. It stores the result into the string object pointed at by *dest*.

This function raises an *ex.AttemptToDerefNULL* exception if *src* or *dest* contain NULL. It raises an *ex.AccessViolation* if *src* or *dest* contain an invalid address. It raises an *ex.StringOverflow* exception if the string object pointed at by *dest* is too small to hold the result.

HLA high-level calling sequence examples:

```
str.upper2( someStr, destStr );

// Do something with destStr
```

HLA low-level calling sequence examples:

```
push( someStr );
push( destStr );
call str.upper2;

// Do something with destStr
```

```
procedure str.a_lower( src:string; dest:string );
```

This function scans the *src* string and converts all upper-case alphabetic characters to their lowercase equivalent. It stores the result into a new string it allocates on the heap (and returns a pointer to this string in EAX). It is the caller's responsibility to free this storage when it is no longer needed.

This function raises an *ex.AttemptToDerefNULL* exception if *src* contains NULL. It raises an *ex.AccessViolation* if *src* contains an invalid address. It raises an *ex.MemoryAllocation* exception if there is an error allocating storage to hold the result.

HLA high-level calling sequence examples:

```
str.a_lower( someStr );
mov( eax, lowerStr );

// Do something with lowerStr
.
.
.
str.free( lowerStr );
```

HLA low-level calling sequence examples:

```
push( someStr );
call str.a_lower;
mov( eax, lowerStr );

// Do something with lowerStr
.
.
.
str.free( lowerStr );
```

```
#macro lower( string );
#macro lower( string, string );
```

This macro provides a "function overload" declaration for the *str.lower1* and *str.lower2* functions. If you pass this macro one argument, it creates a call to the *str.lower1* function; if you pass this macro two arguments, it calls the *str.lower2* function.

```
procedure str.lower1( s:string );
```

This function scans the *s* string and converts, in-place, all upper-case alphabetic characters to their lowercase equivalent.

This function raises an *ex.AttemptToDerefNULL* exception if *s* contains NULL. It raises an *ex.AccessViolation* if *s* contains an invalid address.

HLA high-level calling sequence examples:

```
str.lower1( someStr );

// Do something with someStr
```

HLA low-level calling sequence examples:

```
push( someStr );
call str.lower1;

// Do something with someStr
```

```
procedure str.lower2( src:string; dest:string );
```

This function scans the *src* string and converts all upper-case alphabetic characters to their lowercase equivalent. It stores the result into the string object pointed at by *dest*.

This function raises an *ex.AttemptToDerefNULL* exception if *src* or *dest* contain NULL. It raises an *ex.AccessViolation* if *src* or *dest* contain an invalid address. It raises an *ex.StringOverflow* exception if the string object pointed at by *dest* is too small to hold the result.

HLA high-level calling sequence examples:

```
str.lower2( someStr, destStr );

// Do something with destStr
```

HLA low-level calling sequence examples:

```
push( someStr );
push( destStr );
call str.lower2;

// Do something with destStr
```

```
procedure str.a_reverse( src:string );
```

This function takes the characters in source and creates a new string with the character positions reversed (that is, the first character becomes the last character, the last character becomes the first character, etc.). It stores the result into a new string it allocates on the heap (and returns a pointer to this string in EAX). It is the caller's responsibility to free this storage when it is no longer needed.



This function raises an *ex.AttemptToDerefNULL* exception if *src* contains NULL. It raises an *ex.AccessViolation* if *src* contains an invalid address. It raises an *ex.MemoryAllocation* exception if there is an error allocating storage to hold the result.

HLA high-level calling sequence examples:

```
str.a_reverse( someStr );
mov( eax, reversedStr );

// Do something with reversedStr
.
.
.
str.free( reversedStr );
```

HLA low-level calling sequence examples:

```
push( someStr );
call str.a_reverse;
mov( eax, reversedStr );

// Do something with reversedStr
.
.
.
str.free( reversedStr );
```

```
#macro reverse( string );
#macro reverse( string, string );
```

This macro provides a "function overload" declaration for the *str.reverse1* and *str.reverse2* functions. If you pass this macro one argument, it creates a call to the *str.reverse1* function; if you pass this macro two arguments, it calls the *str.reverse2* function.

```
procedure str.reverse1( s:string );
```

This function scans the *s* string and reverse, in-place, all the characters in the string.

This function raises an *ex.AttemptToDerefNULL* exception if *s* contains NULL. It raises an *ex.AccessViolation* if *s* contains an invalid address.

HLA high-level calling sequence examples:

```
str.reverse1( someStr );

// Do something with someStr
```

HLA low-level calling sequence examples:

```
push( someStr );
call str.reverse1;

// Do something with someStr
```

```
procedure str.reverse2( src:string; dest:string );
```

This function scans the *src* string, reverses their position in the string, storing the result into the *dest* string.

This function raises an *ex.AttemptToDerefNULL* exception if *src* or *dest* contain NULL. It raises an *ex.AccessViolation* if *src* or *dest* contain an invalid address. It raises an *ex.StringOverflow* exception if the string object pointed at by *dest* is too small to hold the result.

HLA high-level calling sequence examples:

```
str.reverse2( someStr, destStr );

// Do something with destStr
```

HLA low-level calling sequence examples:

```
push( someStr );
push( destStr );
call str.reverse2;

// Do something with destStr
```

**procedure str.a\_translate( src:string; from:string; toStr:string );**

This function produces a new string on the heap (and returns a pointer in EAX) by translating all the characters in *src* using the *from* and *toStr* arguments as lookup and translation tables. For each character in *src*, this function scans the *from* string to see if that character is present; if it is not present, the character is copied, untranslated, to the destination string; if the character is present, then the function uses the index of the character in *from* as an index into the *toStr* and fetches that character and outputs it to the destination string. If the index into the *from* string is greater than or equal to the length of the *toStr*, then this function does not copy anything to the destination string (that is, the source character is effectively deleted). It is the caller's responsibility to free up the storage associated with the newly created string when it is no longer needed.

This function raises an *ex.AttemptToDerefNULL* exception if *src*, *from*, or *toStr* contain NULL. It raises an *ex.AccessViolation* if *src*, *from*, or *toStr* contain an invalid address. It raises an *ex.MemoryAllocation* exception if there is an error allocating storage to hold the result.

HLA high-level calling sequence examples:

```
str.a_translate( someStr, lookupStr, conversionStr );
mov( eax, xlatStr );

// Do something with xlatStr
.
.
.
str.free( xlatStr );
```

HLA low-level calling sequence examples:

```
push( someStr );
push( lookupStr );
push( conversionStr );
call str.a_translate;
mov( eax, xlatStr );

// Do something with xlatStr
.
.
.
str.free( xlatStr );
```

```
#macro translate( string, string, string );
#macro translate( string, string, string, string );
```

This macro provides a "function overload" declaration for the *str.translate3* and *str.translate4* functions. If you pass this macro three arguments, it creates a call to the *str.translate3* function; if you pass this macro four arguments, it calls the *str.translate4* function.

```
procedure str.translate3( s:string; from:string; toStr:string );
```

This function modifies the *s* string in-place by translating all the characters in *s* using the *from* and *toStr* arguments as lookup and translation tables. For each character in *s*, this function scans the *from* string to see if that character is present; if it is not present, the character is ignored; if the character is present, then the function uses the index of the character in *from* as an index into the *toStr* and fetches that character and overwrites the original character. If the index into the *from* string is greater than or equal to the length of the *toStr*, then this function deletes that character from the *s* string.

This function raises an *ex.AttemptToDerefNULL* exception if *s*, *from*, or *toStr* contain NULL. It raises an *ex.AccessViolation* if *s*, *from*, or *toStr* contain an invalid address.

HLA high-level calling sequence examples:

```
str.translate3( someStr, fromStr, toStr );

// Do something with someStr
```

HLA low-level calling sequence examples:

```
push( someStr );
push( fromStr );
push( toStr );
call str.translate3;

// Do something with someStr
```

```
procedure str.translate4
(
    src    :string;
    from   :string;
    toStr  :string;
    dest   :string
);
```

This function modifies the *s* by translating all the characters in *s* using the *from* and *toStr* arguments as lookup and translation tables; it stores the modified result into the string object pointed at by *dest*. For each character in *s*, this function scans the *from* string to see if that character is present; if it is not present, the character is simply copied to the destination string; if the character is present in *from*, then the function uses the index of the character in *from* as an index into the *toStr* and fetches that character and copies that to the destination string. If the index into the *from* string is greater than or equal to the length of the *toStr*, then this function does not copy the character to the destination string, effectively deleting it.

This function raises an *ex.AttemptToDerefNULL* exception if *s*, *dest*, *from*, or *toStr* contain NULL. It raises an *ex.AccessViolation* if *s*, *dest*, *from*, or *toStr* contain an invalid address. It raise an *ex.StringOverflow* exception if the string object pointed at by *dest* is too small to hold the result.

HLA high-level calling sequence examples:

```
str.translate4( someStr, fromStr, toStr, dest );
```

```
// Do something with dest
```

HLA low-level calling sequence examples:

```
push( someStr );
push( fromStr );
push( toStr );
push( dest );
call str.translate4;
```

```
// Do something with dest
```

## 33.13 String Concatentation Functions

The functions in this category combine two strings to produce a juxtaposed result.

```
procedure str.a_cat( src1:string; src2:string );
  @returns( "(type string eax)" );
```

This function produces a new string by concatenating the characters in *src1* to the end of the character string specified by *src2*. It stores the result into a new string it allocates on the heap (and returns a pointer to this string in EAX). It is the caller's responsibility to free this storage when it is no longer needed.

This function raises an *ex.AttemptToDerefNULL* exception if *src1* or *src2* contain NULL. It raises an *ex.AccessViolation* if *src1* or *src2* contain an invalid address. It raises an *ex.MemoryAllocation* exception if there is an error allocating storage to hold the result.

HLA high-level calling sequence examples:

```
str.a_cat( "world", "Hello " );
mov( eax, hwStr );

// Do something with "Hello world"
.
.
.
str.free( hwStr );
```

HLA low-level calling sequence examples:

```
static
  helloStr:string := "Hello ";
  worldStr:string := "world";
  .
  .
  .
push( worldStr );
push( helloStr );
call str.a_cat;
mov( eax, hwStr );

// Do something with hwStr
.
.
.
```

```
str.free( hwStr );
```

```
#macro cat( string, string );  
#macro cat( string, string, string );
```

This macro provides a "function overload" declaration for the *str.cat2* and *str.cat3* functions. If you pass this macro two arguments, it creates a call to the *str.cat2* function; if you pass this macro three arguments, it calls the *str.cat3* function.

```
procedure str.cat2( src:string; dest:string );
```

This function produces a new string by concatenating the characters in *src* to the end of *dest*.

This function raises an *ex.AttemptToDerefNULL* exception if *src* or *dest* contain NULL. It raises an *ex.AccessViolation* if *src* or *dest* contain an invalid address. It raises an *ex.StringOverflow* exception if there is insufficient space in the string object pointed at by *dest* to hold the result.

HLA high-level calling sequence examples:

```
str.cat2( someStr, resultStr );  
  
// Do something with resultStr
```

HLA low-level calling sequence examples:

```
push( someStr );  
push( resultStr );  
call str.cat2;  
  
// Do something with resultStr
```

```
procedure str.cat3( src1:string; src2:string; dest:string );
```

This function produces a new string by concatenating the characters in *src1* to the end of the characters in *src2* and storing the result into *dest*.

This function raises an *ex.AttemptToDerefNULL* exception if *src1*, *src2*, or *dest* contain NULL. It raises an *ex.AccessViolation* if *src1*, *src2*, or *dest* contain an invalid address. It raises an *ex.StringOverflow* exception if there is insufficient space in the string object pointed at by *dest* to hold the result.

HLA high-level calling sequence examples:

```
str.cat3( strB, strA, strAB );  
  
// Do something with strAB
```

HLA low-level calling sequence examples:

```
push( strB );  
push( strA );  
push( strAB );  
call str.cat3;  
  
// Do something with strAB
```

```
procedure str.a_catz( src1:zstring; src2:string );
    @returns( "(type string eax)" );
```

This function produces a new HLA string by concatenating the characters in the zero-terminated (zstring) *src1* string to the end of the HLA string specified by *src2*. It stores the result into a new string it allocates on the heap (and returns a pointer to this string in EAX). It is the caller's responsibility to free this storage when it is no longer needed.

This function raises an *ex.AttemptToDerefNULL* exception if *src1* or *src2* contain NULL. It raises an *ex.AccessViolation* if *src1* or *src2* contain an invalid address. It raises an *ex.MemoryAllocation* exception if there is an error allocating storage to hold the result.

HLA high-level calling sequence examples:

```
str.a_catz( zStr, hStr );
mov( eax, h2Str );

// Do something with h2Str
.
.
.
str.free( h2Str );
```

HLA low-level calling sequence examples:

```
static
    helloStr:string := "Hello ";
    worldStr:zstring := "world";
.
.
.
push( worldStr );
push( helloStr );
call str.a_catz;
mov( eax, hwStr );

// Do something with hwStr
.
.
.
str.free( hwStr );
```

```
procedure str.catz( src:zstring; dest:string );
```

This function produces a new string by concatenating the characters in the zero-terminated zstring *src* to the end of the HLA string *dest*.

This function raises an *ex.AttemptToDerefNULL* exception if *src* or *dest* contain NULL. It raises an *ex.AccessViolation* if *src* or *dest* contain an invalid address. It raises an *ex.StringOverflow* exception if there is insufficient space in the string object pointed at by *dest* to hold the result.

HLA high-level calling sequence examples:

```
str.catz( someStr, resultStr );

// Do something with resultStr
```

HLA low-level calling sequence examples:

```
push( someStr );
```

```

push( resultStr );
call str.catz;

// Do something with resultStr

```

```

procedure str.a_catz( src1:string; index:dword; len:dword; src2:string );
    @returns( "(type string eax)" );

```

This function produces a new string by concatenating a substring of *src1* (specified by the *index* and *len* arguments) to the end of the character string specified by *src2*. It stores the result into a new string it allocates on the heap (and returns a pointer to this string in EAX). It is the caller's responsibility to free this storage when it is no longer needed. If *index* is less than or equal to the length of *src1* but the sum of *index+len* is greater than the length of *src1*, then the substring extracted is truncated at the end of the *src1* string.

This function raises an *ex.AttemptToDerefNULL* exception if *src1* or *src2* contain NULL. It raises an *ex.AccessViolation* if *src1* or *src2* contain an invalid address. It raises an *ex.StringIndexError* if *index* is greater than the length of *src1*. It raises an *ex.MemoryAllocation* exception if there is an error allocating storage to hold the result.

HLA high-level calling sequence examples:

```

str.a_cat( "world", "Hello " );
mov( eax, hwStr );

// Do something with "Hello world"
.
.
.
str.free( hwStr );

```

HLA low-level calling sequence examples:

```

static
    helloStr:string := "Hello ";
    worldStr:string := "world";
    .
    .
    .
push( worldStr );
push( helloStr );
call str.a_cat;
mov( eax, hwStr );

// Do something with hwStr
.
.
.
str.free( hwStr );

```

```

#macro catz( string, dword, dword, string );
#macro catz( string, dword, dword, string, string );

```

This macro provides a "function overload" declaration for the *str.cat2* and *str.cat3* functions. If you pass this macro two arguments, it creates a call to the *str.cat2* function; if you pass this macro three arguments, it calls the *str.cat3* function.

```
procedure str.catsub4( src:string; index:dword; len:dword; dest:string );
```

This function produces a new string by concatenating the substr( *src*, *index*, *len* ) to the end of *dest*. If *index* is less than or equal to the length of *src1* but the sum of *index*+*len* is greater than the length of *src1*, then the substring extracted is truncated at the end of the *src1* string.

This function raises an *ex.AttemptToDerefNULL* exception if *src* or *dest* contain NULL. It raises an *ex.AccessViolation* if *src* or *dest* contain an invalid address. It raises an *ex.StringIndexError* exception if *index* is greater than the length of *src*. It raises an *ex.StringOverflow* exception if there is insufficient space in the string object pointed at by *dest* to hold the result.

HLA high-level calling sequence examples:

```
str.catsub4( someStr, 10, 7, resultStr );

// Do something with resultStr
```

HLA low-level calling sequence examples:

```
push( someStr );
pushd( 10 );
pushd( 7 );
push( resultStr );
call str.catsub4;

// Do something with resultStr
```

```
procedure str.catsub5
(
    src1   :string;
    index  :dword;
    len    :dword;
    src2   :string;
    dest   :string
);
```

This function produces a new string by concatenating the characters from substr( *src1*, *index*, *len* ) to the end of the characters in *src2* and storing the result into *dest*. If *index* is less than or equal to the length of *src1* but the sum of *index*+*len* is greater than the length of *src1*, then the substring extracted is truncated at the end of the *src1* string.

This function raises an *ex.AttemptToDerefNULL* exception if *src1*, *src2*, or *dest* contain NULL. It raises an *ex.AccessViolation* if *src1*, *src2*, or *dest* contain an invalid address. It raises an *ex.StringIndexError* exception if *index* is greater than the length of *src1*. It raises an *ex.StringOverflow* exception if there is insufficient space in the string object pointed at by *dest* to hold the result.

HLA high-level calling sequence examples:

```
str.catsub5( rightStr, 10, 7, leftStr, resultStr );

// Do something with resultStr
```

HLA low-level calling sequence examples:

```
push( rightStr );
pushd( 10 );
pushd( 7 );
push( leftStr );
push( resultStr );
call str.catsub5;
```



```
// Do something with resultStr
```

```
procedure str.a_catbuf( startBuf:dword; endBuf:dword; src2:string );
  @returns( "(type string eax)" );
```

This function prototype is just an alias for *str.a\_catbuf3*.

```
procedure str.a_catbuf2( buf:buf_t; src2:string );
  @returns( "(type string eax)" );
```

This function prototype is just an alias for *str.a\_catbuf3*. As it turns out, the *buf* data structure maps exactly to the two dword arguments of *str.a\_catbuf3*.

```
procedure str.a_catbuf3( startBuf:dword; endBuf:dword; src2:string );
  @returns( "(type string eax)" );
```

This function creates a new string on the heap by concatenating the string data from *src2* with the characters found at the address range *startBuf..endBuf* in memory. This function returns a pointer to the new string in the EAX register.

This function raises an *ex.AttemptToDerefNULL* exception if *startBuf*, *endBuf* or *src2* contain NULL. It raises an *ex.AccessViolation* if *startBuf*, *endBuf* or *src2* contain an invalid address. It raises an *ex.StringOverflow* exception if *startBuf* is greater than *endBuf*. It raises an *ex.MemoryAllocation* exception if there is an error allocating storage to hold the result.

HLA high-level calling sequence examples:

```
str.a_catbuf( startPtr, endPtr, "Hello: " );
mov( eax, hwStr );

// Do something with hwStr
.
.
.
str.free( hwStr );
```

HLA low-level calling sequence examples:

```
static
  helloStr:string := "Hello: ";
.
.
.
push( startPtr );
push( endPtr );
push( helloStr );
call str.a_catbuf3;
mov( eax, hwStr );

// Do something with hwStr
.
.
.
str.free( hwStr );
```

```
procedure str.catbuf( startBuf:dword; endBuf:dword; src2:string );
```

This function prototype is just an alias for *str.catbuf3a*.

```
procedure str.catbuf2( buf:buf_t; src2:string );
```

This function prototype is just an alias for *str.a\_catbuf3*. As it turns out, the *buf* data structure maps exactly to the two dword arguments of *str.a\_catbuf3*.

```
procedure str.catbuf3a( startBuf:dword; endBuf:dword; dest:string );
```

This function concatenates the characters in the memory range *startBuf..endBuf* to the end of *dest*.

This function raises an *ex.AttemptToDerefNULL* exception if *startBuf*, *endBuf* or *dest* contain NULL. It raises an *ex.AccessViolation* if *startBuf*, *endBuf* or *dest* contain an invalid address. It raises an *ex.StringOverflow* exception if *startBuf* is greater than *endBuf* or if the resulting string is too large to fit in the string object pointed at by *dest*.

HLA high-level calling sequence examples:

```
str.catbuf3a( startPtr, endPtr, destStr );
```

```
// Do something with destStr
```

HLA low-level calling sequence examples:

```
push( startPtr );
push( endPtr );
push( destStr );
call str.catbuf3;
```

```
// Do something with destStr
```

```
procedure str.catbuf3b( buf:buf_t; src:string; dest:string );
```

This function prototype is just an alias for *str.a\_catbuf4*. As it turns out, the *buf* data structure maps exactly to the two dword arguments of *str.a\_catbuf4*.

```
procedure str.catbuf4
```

```
(
    startBuf    :dword;
    endBuf      :dword;
    src         :string;
    dest        :string
);
```

This function copies the characters in *src* to *dest*, then it concatenates the character in memory (address range *startBuf..endBuf*) to the end of the string in *dest*.

This function raises an *ex.AttemptToDerefNULL* exception if *startBuf*, *endBuf*, *src*, or *dest* contain NULL. It raises an *ex.AccessViolation* if *startBuf*, *endBuf*, *src*, or *dest* contain an invalid address. It raises an *ex.StringOverflow* exception if *startBuf* is greater than *endBuf* or if the resulting string is too large to fit in the string object pointed at by *dest*.

HLA high-level calling sequence examples:

```

    str.catbuf4( startPtr, endPtr, srcStr, destStr );

    // Do something with destStr

HLA low-level calling sequence examples:

push( startPtr );
push( endPtr );
push( srcStr );
push( destStr );
call str.catbuf3;

    // Do something with destStr

```

## 33.14 String Value Concatentation Functions

The functions in this category generally exist to support the *str.put* macro, though they are certainly useful functions in their own right. They convert some data type into string form and concatenate that string to a destination operand.

Note: all of these functions will raise the *ex.StringOverflow* exception if the resulting string does not fit in the destination operation.

Also Note: Functions that do integer/hexadecimal/unsigned numeric conversion may insert underscores between digits, depending on the value of the `stdlib.underscores` flag. See the discussion of `conv.setUnderscores` for more details on this feature.

### 33.14.1 Boolean Output

```

procedure str.catbool( dest:string; b:boolean );

```

This procedure concatenates the string "true" or "false" to the destination string depending on the value of the *b* parameter.

```

HLA high-level calling sequence examples:

str.catbool( dest, boolVar );

// If the boolean is in a register (AL):

str.catbool( dest, al );

HLA low-level calling sequence examples:

    // If "boolVar" is not one of the last three
    // bytes on a page of memory, you can do this:

    push( dest );
    push( (type dword boolVar) );
    call str.catbool;

    // If you can't guarantee that the previous code
    // won't generate an illegal memory access, and a
    // 32-bit register is available, use code like
    // the following:

```

```

push( dest );
movzx( boolVar , eax ); // Assume EAX is available
push( eax );
call str.catbool;

// If no register is available, do something
// like the following code:

push( dest );
sub( 4, esp );
push( eax );
movzx( boolVar , eax );
mov( eax, [esp+4] );
pop( eax );
call str.catbool;

// If the boolean value is in al, bl, cl, or dl
// then you can use code like the following:

push( dest );
push( eax ); // Assume boolVar is in AL
call str.catbool;

// If the Boolean value is in ah, bh, ch, or dh
// you'll have to use code like the following:

push( dest );
xchg( al, ah ); // Assume boolVar is in AH
push( eax );    // It's now in AL
xchg( al, ah ); // Restore al/ah
call str.catbool;

```

### 33.14.2 Character, String, and Character Set Concatenation Routines

**procedure str.catc( dest:string; c:char );**

Appends the character specified by the *c* parameter to the *dest* string.

HLA high-level calling sequence examples:

```
str.catc( dest, charVar );
```

// If the character is in a register (AL):

```
str.catc( dest, al );
```

HLA low-level calling sequence examples:

```
// If "charVar" is not one of the last three
// bytes on a page of memory, you can do this:
```

```

push( dest );
push( (type dword charVar) );
call str.catc;

```

```

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

push( dest );
movzx( charVar, eax ); // Assume EAX is available
push( eax );
call str.catc;

// If no register is available, do something
// like the following code:

push( dest );
sub( 4, esp );
push( eax );
movzx( charVar, eax );
mov( eax, [esp+4] );
pop( eax );
call str.catc;

// If the character value is in al, bl, cl, or dl
// then you can use code like the following:

push( dest );
push( eax ); // Assume charVar is in AL
call str.catc;

// If the character value is in ah, bh, ch, or dh
// you'll have to use code like the following:

push( dest );
xchg( al, ah ); // Assume charVar is in AH
push( eax );    // It's now in AL
xchg( al, ah ); // Restore al/ah
call str.catc;

```

**procedure str.catcSize( dest:string; c:char; width:int32; fill:char )**

Appends the character *c* to the end of *dest* using at least *width* output positions. If the absolute value of *width* is greater than one, then this function emits *fill* characters as padding characters during the concatenation. If *width* is a positive value greater than one, then *str.catcSize* appends *c* left justified in a field of *width* characters; if *width* is a negative value less than one, then *str.catcSize* appends *c* right justified in a field of *width* characters.

HLA high-level calling sequence examples:

```
str.catcSize( dest, charVar, width, padChar );
```

HLA low-level calling sequence examples:

```

// If "charVar" and "padChar" are not one of the last three
// bytes on a page of memory, you can do this:

push( dest );
push( (type dword charVar) );
push( width );
push( (type dword padChar) );
call str.catcSize;

```

```
// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:
```

```
push( dest );
movzx( charVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call str.catcSize;
```

```
// If no registers are available, do something
// like the following code:
```

```
push( dest );
push( eax );
movzx( charVar, eax );
push( eax );
push( width );
movzx( padChar, eax );
push( eax );
call str.catcSize;
pop( eax );
```

```
// If "charVar" or "padChar" are in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:
```

```
push( dest );
push( eax ); // Assume charVar is in AL
push( width );
push( ebx ); // Assume padChar is in BL
call str.catcSize;
```

```
// Do the following if the characters are
// in AH, BH, CH, or DH:
```

```
push( dest );
xchg( al, ah ); // Assume charVar is in AH
xchg( bl, bh ); // Assume padChar is in BH
push( eax );
push( width );
push( ebx );
xchg( al, ah );
xchg( bl, bh );
call str.catcSize;
```

#### **procedure str.catcset( dest:string; cst:cset );**

This function appends a string containing all the members of the *cst* character set parameter to the end of the *dest* string.

HLA high-level calling sequence examples:

```
str.catcset( dest, csVar );
str.catcset( dest, [ebx] ); // EBX points at the cset.
```

HLA low-level calling sequence examples:

```
push( dest );
push( (type dword csVar[12]) ); // Push H.O. dword first
push( (type dword csVar[8]) );
push( (type dword csVar[4]) );
push( (type dword csVar) );      // Push L.O. dword last
call str.catcset;
```

```
push( dest );
push( (type dword [ebx+12]) ); // Push H.O. dword first
push( (type dword [ebx+8]) );
push( (type dword [ebx+4]) );
push( (type dword [ebx]) );    // Push L.O. dword last
call str.catcset;
```

**procedure str.cats( dest:string; s:string );**

This procedure appends the value of the string parameter to the end of the *dest* string. Remember, string values are actually 4-byte pointers to the string's character data. This function is equivalent to the *str.cat* function except that the parameters are reversed to support the *str.put* macro's requirements.

HLA high-level calling sequence examples:

```
str.cats( dest, strVar );
str.cats( dest, ebx ); // EBX holds a string value.
str.cats( dest, "Hello World" );
```

HLA low-level calling sequence examples:

// For string variables:

```
push( dest );
push( strVar );
call str.cats;
```

// For string values held in registers:

```
push( dest );
push( ebx ); // Assume EBX holds the string value
call str.cats;
```

// For string literals, assuming a 32-bit register  
// is available:

```
push( dest );
lea( eax, "Hello World" ); // Assume EAX is available.
push( eax );
call str.cats;
```

// If a 32-bit register is not available:

```
readonly
  literalString :string := "Hello World";
.
```

```

    .
    .
push( dest );
push( literalString );
call str.cats;

```

```
procedure str.catsSize( dest:string; s:string; width:int32; fill:char );
```

This function concatenates the *s* string to the *dest* string using at least *width* character positions. If the absolute value of *width* is less than or equal to the length of *s*, then this function behaves exactly like *str.cats*. On the other hand, if the absolute value of *width* is greater than the length of *s*, then *str.catsSize* appends *width* characters to the *dest* string. This procedure emits the *fill* character in the extra character positions. If *width* is positive, then *str.catsSize* right justifies the string in the output field. If *width* is negative, then *str.catsSize* left justifies the string in the output field. Generally, people expect the string to be left justified, so you should ensure that this value is negative to achieve this.

HLA high-level calling sequence examples:

```

str.catsSize( dest, strVar, width, ' ' );

// For the following, EBX holds the string value,
// ECX contains the width, and AL holds the pad
// character:

str.catsSize( dest, ebx, ecx, al );

str.catsSize( dest, "Hello World", 25, padChar );

```

HLA low-level calling sequence examples:

```

// For string variables:

push( dest );
push( strVar );
push( width );
pushd( ' ' );
call str.catsSize;

// For string values held in registers:

push( dest );
push( ebx ); // Assume EBX holds the string value
push( ecx ); // Assume ECX holds the width
push( eax ); // Assume AL holds the fill character
call str.catsSize;

// For string literals, assuming a 32-bit register
// is available:

push( dest );
lea( eax, "Hello World" ); // Assume EAX is available.
push( eax );
pushd( 25 );
movzx( padChar, eax );
push( eax );
call str.catsSize;

```



```

// If a 32-bit register is not available:

readonly
  literalString :string := "Hello World";

  // Note: element zero is the actual pad character.
  // The other elements are just padding.
  padChar :char[4] := [ '.', #0, #0, #0 ];
  .
  .
  .
push( dest );
push( literalString );
pushd( 25 );
push( (type dword padChar) );
call str.catsSize;

```

### 33.14.3 Hexadecimal Concatenation Routines

These routines convert numeric data to hexadecimal string form (using the hexadecimal conversion routines found in the conv module) and append the result to the destination string.

**procedure str.catb( dest:string; b:byte )**

This procedure appends the value of *b* to the *dest* string using exactly two hexadecimal digits (including a leading zero if necessary).

HLA high-level calling sequence examples:

```

str.catb( dest, byteVar );

// If the character is in a register (AL):

str.catb( dest, al );

```

HLA low-level calling sequence examples:

```

// If "byteVar " is not one of the last three
// bytes on a page of memory, you can do this:

push( dest );
push( (type dword byteVar) );
call str.catb;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

push( dest );
movzx( byteVar, eax ); // Assume EAX is available
push( eax );
call str.catb;

```

```

// If no register is available, do something
// like the following code:

push( dest );
sub( 4, esp );
push( eax );
movzx( byteVar, eax );
mov( eax, [esp+4] );
pop( eax );
call str.catb;

// If the character value is in al, bl, cl, or dl
// then you can use code like the following:

push( dest );
push( eax ); // Assume byteVar is in AL
call str.catb;

// If the byte value is in ah, bh, ch, or dh
// you'll have to use code like the following:

push( dest );
xchg( al, ah ); // Assume byteVar is in AH
push( eax );    // It's now in AL
xchg( al, ah ); // Restore al/ah
call str.catb;

```

**procedure str.cath8( dest:string; b:byte );**

This procedure appends the value of *b* to the *dest* string using the minimum necessary number of hexadecimal digits.

HLA high-level calling sequence examples:

```

str.cath8( dest, byteVar );

// If the character is in a register (AL):

str.cath8( dest, al );

```

HLA low-level calling sequence examples:

```

// If "byteVar " is not one of the last three
// bytes on a page of memory, you can do this:

push( dest );
push( (type dword byteVar) );
call str.cath8;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

```

```

push( dest );
movzx( byteVar , eax ); // Assume EAX is available
push( eax );
call str.cath8;

// If no register is available, do something
// like the following code:

push( dest );
sub( 4, esp );
push( eax );
movzx( byteVar , eax );
mov( eax, [esp+4] );
pop( eax );
call str.cath8;

// If the character value is in al, bl, cl, or dl
// then you can use code like the following:

push( dest );
push( eax ); // Assume byteVar is in AL
call str.cath8;

// If the byte value is in ah, bh, ch, or dh
// you'll have to use code like the following:

push( dest );
xchg( al, ah ); // Assume byteVar is in AH
push( eax );    // It's now in AL
xchg( al, ah ); // Restore al/ah
call str.cath8;

```

**procedure str.cath8Size( dest:string; b:byte; size:dword; fill:char )**

The *str.cath8Size* function concatenates an 8-bit hexadecimal string value to the *dest* string allowing you specify a minimum field *width* and a *fill* character.

HLA high-level calling sequence examples:

```
str.cath8Size( dest, byteVar, width, padChar );
```

HLA low-level calling sequence examples:

```

// If "byteVar" and "padChar" are not one of the last three
// bytes on a page of memory, you can do this:

push( dest );
push( (type dword byteVar) );
push( width );
push( (type dword padChar) );
call str.cath8Size;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

push( dest );

```

```

movzx( byteVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call str.cath8Size;

// If no registers are available, do something
// like the following code:

push( dest );
push( eax );
movzx( byteVar, eax );
push( eax );
push( width );
movzx( padChar, eax );
push( eax );
call str.cath8Size;
pop( eax );

// If "byteVar" or "padChar" are in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:

push( dest );
push( eax ); // Assume byteVar is in AL
push( width );
push( ebx ); // Assume padChar is in BL
call str.cath8Size;

// Do the following if the characters are
// in AH, BH, CH, or DH:

push( dest );
xchg( al, ah ); // Assume byteVar is in AH
xchg( bl, bh ); // Assume padChar is in BH
push( eax );
push( width );
push( ebx );
xchg( al, ah );
xchg( bl, bh );
call str.cath8Size;

```

#### **procedure str.catw( dest:string; w:word )**

This procedure appends the string value of *w* to the *dest* string using exactly four hexadecimal digits (including leading zeros if necessary).

HLA high-level calling sequence examples:

```

str.catw( dest, wordVar );

// If the word is in a register (AX):

str.catw( dest, ax );

```

HLA low-level calling sequence examples:

```
// If "wordVar " is not one of the last three
// bytes on a page of memory, you can do this:

push( dest );
push( (type dword wordVar) );
call str.catw;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

push( dest );
movzx( wordVar, eax ); // Assume EAX is available
push( eax );
call str.catw;

// If no register is available, do something
// like the following code:

push( dest );
push( eax );
movzx( wordVar, eax );
push( eax );
call str.catw;
pop( eax );

// If the word value is in a 16-bit register
// then you can use code like the following:

push( dest );
push( eax ); // Assume wordVar is in AX
call str.catw;
```

#### **procedure str.cath16( dest:string; w:word )**

This procedure appends the string value of *w* to the *dest* string using the minimum necessary number of hexadecimal digits.

HLA high-level calling sequence examples:

```
str.cath16( dest, wordVar );

// If the word is in a register (AX):

str.cath16( dest, ax );
```

HLA low-level calling sequence examples:

```
// If "wordVar " is not one of the last three
```

```

    // bytes on a page of memory, you can do this:

push( dest );
push( (type dword wordVar) );
call str.cath16;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

push( dest );
movzx( wordVar, eax ); // Assume EAX is available
push( eax );
call str.cath16;

// If no register is available, do something
// like the following code:

push( dest );
push( eax );
movzx( wordVar, eax );
push( eax );
call str.cath16;
pop( eax );

// If the word value is in a 16-bit register
// then you can use code like the following:

push( dest );
push( eax ); // Assume wordVar is in AX
call str.cath16;

```

#### **procedure str.cath16Size( dest:string; w:word; size:dword; fill:char )**

The *str.cath16Size* function appends a 16-bit hexadecimal string value to the *dest* string allowing you specify a minimum field width and a fill character.

HLA high-level calling sequence examples:

```
str.cath16Size( dest, wordVar, width, padChar );
```

HLA low-level calling sequence examples:

```

    // If "wordVar" and "padChar" are not one of the last three
    // bytes on a page of memory, you can do this:

push( dest );
push( (type dword wordVar) );
push( width );
push( (type dword padChar) );
call str.cath16Size;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like

```

```
// the following:

push( dest );
movzx( wordVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call str.cath16Size;

// If no registers are available, do something
// like the following code:

push( dest );
push( eax );
movzx( wordVar, eax );
push( eax );
push( width );
movzx( padChar, eax );
push( eax );
call str.cath16Size;
pop( eax );

// If "wordVar" is in a 16-bit register
// and "padChar" is in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:

push( dest );
push( eax ); // Assume wordVar is in AX
push( width );
push( ebx ); // Assume padChar is in BL
call str.cath16Size;
```

#### **procedure str.catd( dest:string; d:dword )**

This procedure appends the string value of *d* to the *dest* string using exactly eight hexadecimal digits (including leading zeros if necessary), if underscore output is not enabled. This routine will emit nine characters (eight digits plus an underscore) if underscore output is enabled.

HLA high-level calling sequence examples:

```
str.catd( dest, dwordVar );

// If the dword value is in a register (EAX):

str.catd( dest, eax );
```

HLA low-level calling sequence examples:

```
push( dest );
push( dwordVar );
call str.catd;

push( dest );
push( eax );
call str.catd;
```

```
procedure str.cath32( dest:string; d:dword );
```

This procedure appends the string value of *d* to the *dest* string using the minimum number of hexadecimal digits necessary. If underscore output is enabled (see `conv.setUnderscores` and `conv.getUnderscores`) then this function will emit an underscore between groups of four hexadecimal digits, starting from the least significant digit.

HLA high-level calling sequence examples:

```
str.cath32( dest, dwordVar );

// If the dword is in a register (EAX):

str.cath32( dest, eax );
```

HLA low-level calling sequence examples:

```
push( dest );
push( dwordVar );
call str.cath32;

push( dest );
push( eax );
call str.cath32;
```

```
procedure str.cath32Size( dest:string; d:dword; size:dword; fill:char )
```

The *str.cath32Size* function outputs *d* as a hexadecimal string to the end of the *dest* string (including underscores, if enabled) and it allows you specify a minimum field *width* and a *fill* character.

HLA high-level calling sequence examples:

```
str.cath32Size( dest, dwordVar, width, ' ' );

// If the dword is in a register (EAX):

str.cath32Size( dest, eax, width, cl );
```

HLA low-level calling sequence examples:

```
push( dest );
push( dwordVar );
push( width );
pushd( ' ' );
call str.cath32Size;

push( dest );
push( eax );
push( width );
push( ecx ); // fill char is in CL
call str.cath32Size;

// Assume fill char is in CH
```



```

push( dest );
push( eax );
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call str.cath32Size;

// Alternate method of the above

push( dest );
push( eax );
push( width );
sub( 4, esp );
mov( ch, [esp] );
call str.cath32Size;

// If the fill char is a variable and
// a register is available, try this code:

push( dest );
push( eax );
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call str.cath32Size;

// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( dest );
push( eax );
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call str.cath32Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( dest );
push( eax );
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call str.cath32Size;

```

**procedure str.catq( dest:string; q:qword );**

This procedure appends the value of *q* to the *dest* string using exactly sixteen hexadecimal digits (including leading zeros if necessary and an intervening underscores if underscore output is enabled).

HLA high-level calling sequence examples:

```
str.catq( dest, qwordVar );
```

HLA low-level calling sequence examples:

```
push( dest );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
call str.catq;
```

**procedure str.cath64( dest:string; q:qword );**

This procedure appends the value of *q* to the *dest* string using the minimum necessary number of hexadecimal digits (including intervening underscores if underscore output is enabled).

HLA high-level calling sequence examples:

```
str.cath64( dest, qwordVar );
```

HLA low-level calling sequence examples:

```
push( dest );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
call str.cath64;
```

**procedure str.cath64Size( dest:string; q:qword; size:dword; fill:char );**

The *str.cath64Size* function lets you specify a minimum field width and a fill character when appending the string form of the *q* parameter to the end of the *dest* string. Note that if underscore output is enabled, this routine will emit up to 19 characters (16 digits plus three underscores).

HLA high-level calling sequence examples:

```
str.cath64Size( dest, qwordVar, width, ' ' );
```

HLA low-level calling sequence examples:

```
push( dest );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
pushd( ' ' );
call str.cath64Size;
```

```
push( dest );
push( edx ); // Assume 64-bit value in edx:eax
push( eax );
push( width );
push( ecx ); // fill char is in CL
call str.cath64Size;
```

```

// Assume fill char is in CH

push( dest );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));      // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call str.cath64Size;

// Alternate method of the above

push( dest );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));      // L.O. dword last
push( width );
sub( 4, esp );
mov( ch, [esp] );
call str.cath64Size;

// If the fill char is a variable and
// a register is available, try this code:

push( dest );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));      // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call str.cath64Size;

// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( dest );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));      // L.O. dword last
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call str.cath64Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( dest );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));      // L.O. dword last
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call str.cath64Size;

```

```
procedure str.cattb( dest:string; tb:tbyte );
```

This procedure appends the string value of *tb* to the *dest* string using exactly 20 hexadecimal digits (including leading zeros if necessary and an intervening underscores if underscore output is enabled).

HLA high-level calling sequence examples:

```
str.cattb( dest, tbyteVar );
```

HLA low-level calling sequence examples:

```
push( dest );
pushw( 0 );           // Push push a 0 pad word
push( (type word tbyteVar[8])); // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));    // L.O. dword last
call str.cattb;
```

```
procedure str.cath80( dest:string; tb:tbyte );
```

This procedure appends the value of *tb* to the *dest* string using the minimum necessary number of hexadecimal digits (including intervening underscores if underscore output is enabled).

HLA high-level calling sequence examples:

```
str.cath80( dest, tbyteVar );
```

HLA low-level calling sequence examples:

```
push( dest );
pushw( 0 );           // Push push a 0 pad word
push( (type word tbyteVar[8])); // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));    // L.O. dword last
call str.cath80;
```

```
procedure str.cath80Size( dest:string; tb:tbyte; width:dword; fill:char );
```

The *str.cath80Size* function appends the hexadecimal form of the 80-bit *tb* parameter to the end of the *dest* string. It lets you specify a minimum field *width* and a *fill* character.

HLA high-level calling sequence examples:

```
str.cath80Size( dest, tbyteVar, width, ' ' );
```

HLA low-level calling sequence examples:

```
push( dest );
pushw( 0 );           // Push push a 0 pad word
push( (type word tbyteVar[8])); // Push H.O. word first
```

```

push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));    // L.O. dword last
push( width );
pushd( ' ' );
call str.cath80Size;

// Assume fill char is in CH

push( dest );
pushw( 0 );                      // Push push a 0 pad word
push( (type word tbyteVar[8]));  // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));    // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call str.cath80Size;

// Alternate method of the above

push( dest );
pushw( 0 );                      // Push push a 0 pad word
push( (type word tbyteVar[8]));  // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));    // L.O. dword last
push( width );
sub( 4, esp );
mov( ch, [esp] );
call str.cath80Size;

// If the fill char is a variable and
// a register is available, try this code:

push( dest );
pushw( 0 );                      // Push push a 0 pad word
push( (type word tbyteVar[8]));  // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));    // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call str.cath80Size;

// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( dest );
pushw( 0 );                      // Push push a 0 pad word
push( (type word tbyteVar[8]));  // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar));    // L.O. dword last
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call str.cath80Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

```

```

push( dest );
pushw( 0 );           // Push push a 0 pad word
push( (type word tbyteVar[8]) ); // Push H.O. word first
push( (type dword tbyteVar[4]) ); // M.O. dword second
push( (type dword tbyteVar) );    // L.O. dword last
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call str.cath80Size;

```

**procedure str.cat1( dest:string; l:lword );**

This procedure appends the string value of *l* to the *dest* string using exactly 32 hexadecimal digits (including leading zeros if necessary and intervening underscores if underscore output is enabled).

HLA high-level calling sequence examples:

```
str.cat1( dest, lwordVar );
```

HLA low-level calling sequence examples:

```

push( dest );
push( (type dword lwordVar[12]) ); // H.O. dword first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar) );    // L.O. dword last
call str.cat1;

```

**procedure str.cath128( dest:string; l:lword );**

This procedure appends the string value of *l* to the *dest* string using the minimum necessary number of hexadecimal digits (including intervening underscores if underscore output is enabled).

HLA high-level calling sequence examples:

```
str.cath128( dest, lwordVar );
```

HLA low-level calling sequence examples:

```

push( dest );
push( (type dword lwordVar[12]) ); // H.O. dword first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar) );    // L.O. dword last
call str.cath128;

```

**procedure str.cath128Size( dest:string; l:lword; width:dword; fill:char );**

The *str.cath128Size* function appends the string value of *l* to the *dest* string and it lets you specify a minimum field *width* and a *fill* character.

HLA high-level calling sequence examples:

```
str.cath128Size( dest, tbyteVar, width, ' ' );
```

HLA low-level calling sequence examples:

```
push( dest );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));    // L.O. dword last
push( width );
pushd( ' ' );
call str.cath128Size;
```

```
// Assume fill char is in CH
```

```
push( dest );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));    // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call str.cath128Size;
```

```
// Alternate method of the above
```

```
push( dest );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));    // L.O. dword last
push( width );
sub( 4, esp );
mov( ch, [esp] );
call str.cath128Size;
```

```
// If the fill char is a variable and
// a register is available, try this code:
```

```
push( dest );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));    // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call str.cath128Size;
```

```

// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( dest );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
push( width );
push( (type dword fillChar) );     // Chance of page crossing!
call str.cath128Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( dest );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call str.cath128Size;

```

### 33.14.4 Signed Integer Concatenation Routines

These routines convert signed integer values to string format and append that string to a destination string. The *str.catxxxSize* functions contain *width* and *fill* parameters that let you specify the minimum field width when outputting a value.

If the absolute value of *width* is greater than the number of print positions the value requires, then these functions output *width* characters to the string. If *width* is non-negative, then these functions right-justify the value in the output field; if *width* is negative, then these functions left-justify the value in the output field.

These functions print the *fill* character as the padding value for the extra output positions.

Note that unlike floating point values, these functions do not print a space in front of the value if it is non-negative.

**xxxSize( value, width, fill );**

Assuming “value” requires five print positions, “width” is eight, and fill is “f” then the xxxSize functions produce the string

f	f	f	V	A	L	U	E
---	---	---	---	---	---	---	---

Assuming “value” requires five print positions, “width” is minus eight, and fill is “f” then the xxxSize functions produce the string

V	A	L	U	E	f	f	f
---	---	---	---	---	---	---	---

**procedure str.cati8 ( dest:string; b:byte );**

This function converts the eight-bit signed integer you pass as a parameter to a string and appends this string to *dest* using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:



```

str.cati8( dest, byteVar );

// If the character is in a register (AL):

str.cati8( dest, al );

```

HLA low-level calling sequence examples:

```

// If "byteVar " is not one of the last three
// bytes on a page of memory, you can do this:

push( dest );
push( (type dword byteVar ) );
call str.cati8;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

push( dest );
movzx( byteVar , eax ); // Assume EAX is available
push( eax );
call str.cati8;

// If no register is available, do something
// like the following code:

push( dest );
push( eax );
movzx( byteVar , eax );
push( eax );
call str.cati8;
pop( eax );

// If the character value is in al, bl, cl, or dl
// then you can use code like the following:

push( dest );
push( eax ); // Assume byteVar is in AL
call str.cati8;

// If the byte value is in ah, bh, ch, or dh
// you'll have to use code like the following:

push( dest );
xchg( al, ah ); // Assume byteVar is in AH
push( eax );    // It's now in AL
xchg( al, ah ); // Restore al/ah
call str.cati8;

```

```
procedure str.cati8Size ( dest:string; b:byte; width:int32; fill:char )
```

This function appends the eight-bit signed integer value you pass to the *dest* string using the *width* and *fill* values as specified above.

HLA high-level calling sequence examples:

```
str.cati8Size( dest, byteVar, width, padChar );
```

HLA low-level calling sequence examples:

```
// If "byteVar" and "padChar" are not one of the last three
// bytes on a page of memory, you can do this:
```

```
push( dest );
push( (type dword byteVar) );
push( width );
push( (type dword padChar) );
call str.cati8Size;
```

```
// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:
```

```
push( dest );
movzx( byteVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call str.cati8Size;
```

```
// If no registers are available, do something
// like the following code:
```

```
push( dest );
push( eax );
movzx( byteVar, eax );
push( eax );
push( width );
movzx( padChar, eax );
push( eax );
call str.cati8Size;
pop( eax );
```

```
// If "byteVar" or "padChar" are in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:
```

```
push( dest );
push( eax ); // Assume byteVar is in AL
push( width );
push( ebx ); // Assume padChar is in BL
call str.cati8Size;
```

```
// Do the following if the characters are
// in AH, BH, CH, or DH:
```

```
push( dest );
```

```

xchg( al, ah );    // Assume byteVar is in AH
xchg( bl, bh );    // Assume padChar is in BH
push( eax );
push( width );
push( ebx );
xchg( al, ah );
xchg( bl, bh );
call str.cat18Size;

```

**procedure str.cat16( dest:string; w:word );**

This function converts the 16-bit signed integer you pass as a parameter to a string and append this string to *dest* using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```

str.cat16( dest, wordVar );

// If the word is in a register (AX):

str.cat16( dest, ax );

```

HLA low-level calling sequence examples:

```

// If "wordVar " is not one of the last three
// bytes on a page of memory, you can do this:

push( dest );
push( (type dword wordVar) );
call str.cat16;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

push( dest );
movzx( wordVar, eax ); // Assume EAX is available
push( eax );
call str.cat16;

// If no register is available, do something
// like the following code:

push( dest );
push( eax );
movzx( wordVar, eax );
push( eax );
call str.cat16;
pop( eax );

// If the word value is in a 16-bit register
// then you can use code like the following:

push( dest );

```

```
push( eax ); // Assume wordVar is in AX
call str.catil6;
```

**procedure str.catil6Size( dest:string; w:word; width:int32; fill:char );**

This function appends the 16-bit signed integer value you pass to the *dest* string using the *width* and *fill* values as specified above.

HLA high-level calling sequence examples:

```
str.catil6Size( dest, wordVar, width, padChar );
```

HLA low-level calling sequence examples:

```
// If "wordVar" and "padChar" are not one of the last three
// bytes on a page of memory, you can do this:
```

```
push( dest );
push( (type dword wordVar) );
push( width );
push( (type dword padChar) );
call str.catil6Size;
```

```
// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:
```

```
push( dest );
movzx( wordVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call str.catil6Size;
```

```
// If no registers are available, do something
// like the following code:
```

```
push( dest );
push( eax );
movzx( wordVar, eax );
push( eax );
push( width );
movzx( padChar, eax );
push( eax );
call str.catil6Size;
pop( eax );
```

```
// If "wordVar" is in a 16-bit register
// and "padChar" is in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:
```

```
push( dest );
```

```

push( eax );    // Assume wordVar is in AX
push( width );
push( ebx );    // Assume padChar is in BL
call str.cat16Size;

```

**procedure str.cat132( dest:string; d:dword );**

This function converts the 32-bit signed integer you pass as a parameter to a string and appends it to *dest* using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```

str.cat132( dest, dwordVar );

// If the dword is in a register (EAX):

str.cat132( dest, eax );

```

HLA low-level calling sequence examples:

```

push( dest );
push( dwordVar );
call str.cat132;

push( dest );
push( eax );
call str.cat132;

```

**procedure str.cat132Size( dest:string; d:dword; width:int32; fill:char );**

This function appends the 32-bit value you pass as a signed integer to the *dest* string using the *width* and *fill* values as specified above.

HLA high-level calling sequence examples:

```

str.catu32Size( dest, dwordVar, width, ' ' );

// If the dword is in a register (EAX):

str.catu32Size( dest, eax, width, cl );

```

HLA low-level calling sequence examples:

```

push( dest );
push( dwordVar );
push( width );
pushd( ' ' );
call str.catu32Size;

push( dest );
push( eax );
push( width );

```

```

push( ecx ); // fill char is in CL
call str.catu32Size;

// Assume fill char is in CH

push( dest );
push( eax );
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call str.catu32Size;

// Alternate method of the above

push( dest );
push( eax );
push( width );
sub( 4, esp );
mov( ch, [esp] );
call str.catu32Size;

// If the fill char is a variable and
// a register is available, try this code:

push( dest );
push( eax );
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call str.catu32Size;

// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( dest );
push( eax );
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call str.catu32Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( dest );
push( eax );
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call str.cati32Size;

```

```
procedure str.cati64( dest:string; q:qword );
```

This function converts the 64-bit signed integer you pass as a parameter to a string and appends this string to *dest* using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
str.cati64( dest, qwordVar );
```

HLA low-level calling sequence examples:

```
push( dest );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
call str.cati64;
```

```
procedure str.cati64Size( dest:string; q:qword; width:int32; fill:char );
```

This function appends the 64-bit value you pass as a signed integer to the *dest* string using the *width* and *fill* values as specified earlier.

HLA high-level calling sequence examples:

```
str.cati64Size( dest, qwordVar, width, ' ' );
```

HLA low-level calling sequence examples:

```
push( dest );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
pushd( ' ' );
call str.cati64Size;
```

```
push( dest );
push( edx ); // Assume 64-bit value in edx:eax
push( eax );
push( width );
push( ecx ); // fill char is in CL
call str.cati64Size;
```

```
// Assume fill char is in CH
```

```
push( dest );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call str.cati64Size;
```

```
// Alternate method of the above
```

```

push( dest );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));      // L.O. dword last
push( width );
sub( 4, esp );
mov( ch, [esp] );
call str.cati64Size;

// If the fill char is a variable and
// a register is available, try this code:

push( dest );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));      // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call str.cati64Size;

// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( dest );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));      // L.O. dword last
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call str.cati64Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( dest );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));      // L.O. dword last
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call str.cati64Size;

```

**procedure str.cati128( dest:string; l:lword );**

This function converts the 128-bit signed integer you pass as a parameter to a string and appends this string to *dest* using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
str.cati128( dest, lwordVar );
```

HLA low-level calling sequence examples:



```

push( dest );
push( (type dword lwordVar[12]) ); // H.O. dword first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
call str.catil28;

```

**procedure str.catil28Size( dest:string; l:lword; width:int32; fill:char );**

This function appends the 128-bit value you pass as a signed integer to the *dest* string using the *width* and *fill* values as specified above.

HLA high-level calling sequence examples:

```
str.catil28Size( dest, lwordVar, width, ' ' );
```

HLA low-level calling sequence examples:

```

push( dest );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
push( width );
pushd( ' ' );
call str.catil28Size;

```

// Assume fill char is in CH

```

push( dest );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call str.catil28Size;

```

// Alternate method of the above

```

push( dest );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
push( width );
sub( 4, esp );
mov( ch, [esp] );
call str.catil28Size;

```

// If the fill char is a variable and  
// a register is available, try this code:

```
push( dest );
```

```

push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar)); // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call str.cat128Size;

// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( dest );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar)); // L.O. dword last
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call str.cat128Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( dest );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar)); // L.O. dword last
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call str.cat128Size;

```

### 33.14.5 Unsigned Integer Concatenation Routines

These routines convert unsigned integer values to string format and append that string to the destination string passed as an argument. The *str.catxxxSize* functions contain *width* and *fill* parameters that let you specify the minimum field width when outputting a value.

If the absolute value of *width* is greater than the number of character positions the value requires, then these functions append *width* characters to the destination string. If *width* is non-negative, then these functions right-justify the value in the output field; if *width* is negative, then these functions left-justify the value in the output field.

These functions emit the *fill* character as the padding value for the extra print positions.

```
xxxSize( value, width, fill );
```

Assuming “value” requires five print positions, “width” is eight, and fill is “f” then the xxxSize functions produce the string

f	f	f	V	A	L	U	E
---	---	---	---	---	---	---	---

Assuming “value” requires five print positions, “width” is minus eight, and fill is “f” then the xxxSize functions produce the string

V	A	L	U	E	f	f	f
---	---	---	---	---	---	---	---

```
procedure str.catu8 ( dest:string; b:byte );
```

This function converts the eight-bit unsigned integer you pass as a parameter to a string and appends this string to *dest* using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```
str.catu8( dest, byteVar );
```

```
// If the character is in a register (AL):
```

```
str.catu8( dest, al );
```

HLA low-level calling sequence examples:

```
// If "byteVar " is not one of the last three
// bytes on a page of memory, you can do this:
```

```
push( dest );
push( (type dword byteVar ) );
call str.catu8;
```

```
// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:
```

```
push( dest );
movzx( byteVar , eax ); // Assume EAX is available
push( eax );
call str.catu8;
```

```
// If no register is available, do something
// like the following code:
```

```
push( dest );
push( eax );
movzx( byteVar , eax );
push( eax );
call str.catu8;
pop( eax );
```

```
// If the character value is in al, bl, cl, or dl
// then you can use code like the following:
```

```

push( dest );
push( eax ); // Assume byteVar is in AL
call str.catu8;

// If the byte value is in ah, bh, ch, or dh
// you'll have to use code like the following:

push( dest );
xchg( al, ah ); // Assume byteVar is in AH
push( eax );    // It's now in AL
xchg( al, ah ); // Restore al/ah
call str.catu8;

```

**procedure str.catu8Size( dest:string; b:byte; width:int32; fill:char );**

This function appends the unsigned eight-bit value you pass to the *dest* string using the *width* and *fill* values as specified above.

HLA high-level calling sequence examples:

```
str.catu8Size( dest, byteVar, width, padChar );
```

HLA low-level calling sequence examples:

```

// If "byteVar" and "padChar" are not one of the last three
// bytes on a page of memory, you can do this:

push( dest );
push( (type dword byteVar) );
push( width );
push( (type dword padChar) );
call str.catu8Size;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

push( dest );
movzx( byteVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call str.catu8Size;

// If no registers are available, do something
// like the following code:

push( dest );
push( eax );
movzx( byteVar, eax );
push( eax );
push( width );
movzx( padChar, eax );
push( eax );
call str.catu8Size;
pop( eax );

```

```

// If "byteVar" or "padChar" are in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:

push( dest );
push( eax );    // Assume byteVar is in AL
push( width );
push( ebx );    // Assume padChar is in BL
call str.catu8Size;

// Do the following if the characters are
// in AH, BH, CH, or DH:

push( dest );
xchg( al, ah );    // Assume byteVar is in AH
xchg( bl, bh );    // Assume padChar is in BH
push( eax );
push( width );
push( ebx );
xchg( al, ah );
xchg( bl, bh );
call str.catu8Size;

```

**procedure str.catu16( dest:string; w:word );**

This function converts the 16-bit unsigned integer you pass as a parameter to a string and appends this to the *dest* string using the minimum number of print positions the number requires.

HLA high-level calling sequence examples:

```

str.catu16( dest, wordVar );

// If the word is in a register (AX):

str.catu16( dest, ax );

```

HLA low-level calling sequence examples:

```

// If "wordVar " is not one of the last three
// bytes on a page of memory, you can do this:

push( dest );
push( (type dword wordVar) );
call str.catu16;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

push( dest );
movzx( wordVar, eax ); // Assume EAX is available

```

```

push( eax );
call str.catu16;

// If no register is available, do something
// like the following code:

push( dest );
push( eax );
movzx( wordVar, eax );
push( eax );
call str.catu16;
pop( eax );

// If the word value is in a 16-bit register
// then you can use code like the following:

push( dest );
push( eax ); // Assume wordVar is in AX
call str.catu16;

```

**procedure str.catu16Size( dest:string; w:word; width:int32; fill:char );**

This function appends the unsigned 16-bit value you pass to the *dest* string using the *width* and *fill* values as specified above.

HLA high-level calling sequence examples:

```
str.catu16Size( dest, wordVar, width, padChar );
```

HLA low-level calling sequence examples:

```

// If "wordVar" and "padChar" are not one of the last three
// bytes on a page of memory, you can do this:

push( dest );
push( (type dword wordVar) );
push( width );
push( (type dword padChar) );
call str.catu16Size;

// If you can't guarantee that the previous code
// won't generate an illegal memory access, and a
// 32-bit register is available, use code like
// the following:

push( dest );
movzx( wordVar, eax ); // Assume EAX is available
push( eax );
push( width );
movzx( padChar, ebx ); // Assume EBX is available
push( ebx );
call str.catu16Size;

// If no registers are available, do something
// like the following code:

push( dest );
push( eax );

```

```

movzx( wordVar, eax );
push( eax );
push( width );
movzx( padChar, eax );
push( eax );
call str.catu16Size;
pop( eax );

// If "wordVar" is in a 16-bit register
// and "padChar" is in an
// 8-bit register, then you can push
// the corresponding 32-bit register if
// the register is AL, BL, CL, or DL:

push( dest );
push( eax );    // Assume wordVar is in AX
push( width );
push( ebx );    // Assume padChar is in BL
call str.catu16Size;

```

**procedure str.catu32( dest:string; d:dword );**

This function converts the 32-bit unsigned integer you pass as a parameter to a string and appends this to the *dest* string using the minimum number of character positions the number requires.

HLA high-level calling sequence examples:

```

str.catu32( dest, dwordVar );

// If the dword is in a register (EAX):

str.catu32( dest, eax );

```

HLA low-level calling sequence examples:

```

push( dest );
push( dwordVar );
call str.catu32;

push( dest );
push( eax );
call str.catu32;

```

**procedure str.catu32Size( dest:string; d:dword; width:int32; fill:char );**

This function appends the unsigned 32-bit value you pass to the *dest* string using the *width* and *fill* values as specified above.

HLA high-level calling sequence examples:

```

str.catu32Size( dest, dwordVar, width, ' ' );

```

```

// If the dword is in a register (EAX):

str.catu32Size( dest, eax, width, cl );


HLA low-level calling sequence examples:


push( dest );
push( dwordVar );
push( width );
pushd( ' ' );
call str.catu32Size;


push( dest );
push( eax );
push( width );
push( ecx ); // fill char is in CL
call str.catu32Size;


// Assume fill char is in CH

push( dest );
push( eax );
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call str.catu32Size;


// Alternate method of the above

push( dest );
push( eax );
push( width );
sub( 4, esp );
mov( ch, [esp] );
call str.catu32Size;


// If the fill char is a variable and
// a register is available, try this code:

push( dest );
push( eax );
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call str.catu32Size;


// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( dest );
push( eax );
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call str.catu32Size;


// In the very rare case that the above would

```



```
// cause an illegal memory access, use this:
```

```
push( dest );
push( eax );
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call str.catu32Size;
```

**procedure str.catu64( dest:string; q:qword );**

This function converts the 64-bit unsigned integer you pass as a parameter to a string and appends this string to *dest* using the minimum number of character positions the integer requires.

HLA high-level calling sequence examples:

```
str.catu64( dest, qwordVar );
```

HLA low-level calling sequence examples:

```
push( dest );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
call str.catu64;
```

**procedure str.catu64Size( dest:string; q:qword; width:int32; fill:char );**

This function appends the unsigned 64-bit value you pass to the *dest* string using the *width* and *fill* values as specified above.

HLA high-level calling sequence examples:

```
str.catu64Size( dest, qwordVar, width, ' ' );
```

HLA low-level calling sequence examples:

```
push( dest );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
pushd( ' ' );
call str.catu64Size;
```

```
push( dest );
push( edx ); // Assume 64-bit value in edx:eax
push( eax );
push( width );
push( ecx ); // fill char is in CL
call str.catu64Size;
```

```
// Assume fill char is in CH
```

```

push( dest );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call str.catu64Size;

// Alternate method of the above

push( dest );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
sub( 4, esp );
mov( ch, [esp] );
call str.catu64Size;

// If the fill char is a variable and
// a register is available, try this code:

push( dest );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call str.catu64Size;

// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( dest );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
push( (type dword fillChar) ); // Chance of page crossing!
call str.catu64Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( dest );
push( (type dword qwordVar[4]) ); // H.O. dword first
push( (type dword qwordVar));    // L.O. dword last
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call str.catu64Size;

```

**procedure str.catu128( dest:string; l:lword );**

This function converts the 128-bit unsigned integer you pass as a parameter to a string and appends this string to *dest* using the minimum number of character positions the number requires.

HLA high-level calling sequence examples:

```
str.catu128( dest, lwordVar );
```

HLA low-level calling sequence examples:

```
push( dest );
push( (type dword lwordVar[12]) ); // H.O. dword first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
call str.catu128;
```

**procedure str.catu128Size( dest:string; l:lword; width:int32; fill:char )**

This function appends the unsigned 128-bit value you pass to the *dest* string using the *width* and *fill* values as specified above.

HLA high-level calling sequence examples:

```
str.catu128Size( dest, lwordVar, width, ' ' );
```

HLA low-level calling sequence examples:

```
push( dest );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
push( width );
pushd( ' ' );
call str.catu128Size;
```

```
// Assume fill char is in CH
```

```
push( dest );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));      // L.O. dword last
push( width );
xchg( cl, ch ); // fill char is in CH
push( ecx );
xchg( cl, ch );
call str.catu128Size;
```

```
// Alternate method of the above
```

```

push( dest );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));    // L.O. dword last
push( width );
sub( 4, esp );
mov( ch, [esp] );
call str.catu128Size;

// If the fill char is a variable and
// a register is available, try this code:

push( dest );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));    // L.O. dword last
push( width );
movzx( fillChar, ebx ); // Assume EBX is available
push( ebx );
call str.catu128Size;

// If the fill char is a variable and
// no register is available, here's one
// possibility:

push( dest );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));    // L.O. dword last
push( width );
push( (type dword fillChar) );    // Chance of page crossing!
call str.catu128Size;

// In the very rare case that the above would
// cause an illegal memory access, use this:

push( dest );
push( (type dword lwordVar[12])); // Push H.O. word first
push( (type dword lwordVar[8]) );
push( (type dword lwordVar[4]) );
push( (type dword lwordVar));    // L.O. dword last
push( width );
sub( 4, esp );
push( eax );
movzx( fillChar, eax );
mov( eax, [esp+4] );
pop( eax );
call str.catu128Size;

```

## 33.15 Floating-Point Concatenation Routines

The HLA string module provides several procedures you can use to append the string representation of floating point values to some string. The following subsections describe these routines.

### 33.15.1 Real to String Output Using Scientific Notation

The floating point numeric output routines translate the three different binary floating point formats to their string representation and then concatenate this string to some destination string. There are two generic classes of these routines: those that convert their values to exponential/scientific notation and those that convert their string to a decimal form.

The *str.cate80*, *str.cate64*, and *str.cate32* routines convert their values to a string using scientific notation. These three routines each have two parameters: the value to output and the field width of the result. These routines produce a string with the following format:

s	i	.	f	f	f	f	f	E	±	x
---	---	---	---	---	---	---	---	---	---	---

s is a space for positive values, '-' for negative values  
i represents the integer portion of the mantissa  
ffff represents the fractional portion of the mantissa  
x is one or more base-10 exponent digits.

```
procedure str.cate32( dest:string; r:real32; width:uns32 );
```

This function appends string conversion of the 32-bit single precision floating point value passed in *r* to the *dest* string using scientific/exponential notation. This procedure prints the value using *width* print positions in the output. *width* should have a minimum value of of six. Note that 32-bit extended precision floating point values support about 6-7 significant digits. So a *width* value that yields more than seven mantissa digits will produce garbage output in the low order digits of the number.

HLA high-level calling sequence examples:

```
str.cate32( dest, r32Var, width );

// If the real32 value is in an FPU register (ST0):

var
    r32Temp:real32;
.
.
.
fstp( r32Temp );
str.cate32( dest, r32Temp, 12 );
```

HLA low-level calling sequence examples:

```
push( dest );
push( (type dword r32Var) );
push( width );
call str.cate32;

push( dest );
sub( 4, esp );
fstp( (type real32 [esp]) );
pushd( 12 );
call str.cate32;
```

```
procedure str.cate64( dest:string; r:real64; width:uns32 );
```

This function appends the string conversion of the 64-bit double precision floating point value passed in *r* to the *dest* string using scientific/exponential notation. This procedure appends the value using *width* character positions in the output. *width* should have a minimum value of six. Note that 64-bit double precision floating point values support about 15 significant digits. So a *width* value that yeilds more than 15 mantissa digits will produce garbage output in the low order digits of the number.

HLA high-level calling sequence examples:

```
str.cate64( dest, r64Var, width );

// If the real64 value is in an FPU register (ST0):

var
    r64Temp:real64;
    .
    .
    .
fstp( r64Temp );
str.cate64( dest, r64Temp, 12 );
```

HLA low-level calling sequence examples:

```
push( dest );
push( (type dword r64Var[4]));
push( (type dword r64Var[0]));
push( width );
call str.cate64;

push( dest );
sub( 8, esp );
fstp( (type real64 [esp]) );
pushd( 12 );
call str.cate64;
```

```
procedure str.cate80( dest:string; r:real80; width:uns32 );
```

This function appends the string conversion of the 80-bit extended precision floating point value passed in *r* to the *dest* string using scientific/exponential notation. This procedure emits the value using *width* character positions in *dest*. *width* should have a minimum value of six. Note that 80-bit extended precision floating point values support about 18 significant digits. So a *width* value that yeilds more than 18 mantissa digits will produce garbage output in the low order digits of the number.

HLA high-level calling sequence examples:

```
str.cate80( dest, r80Var, width );

// If the real80 value is in an FPU register (ST0):

var
    r80Temp:real80;
    .
    .
    .
```

```

    fstp( r80Temp );
    str.cate80( dest, r80Temp, 12 );

```

HLA low-level calling sequence examples:

```

push( dest );
pushw( 0 ); // A word of padding.
push( (type word r80Var[8]) );
push( (type dword r80Var[4]) );
push( (type dword r80Var[0]) );
push( width );
call str.cate80;

```

```

push( dest );
sub( 12, esp );
fstp( (type real80 [esp]) );
pushd( 12 );
call str.cate80;

```

### 33.15.2 Real To String Output Using Decimal Notation

Although scientific (exponential) notation is the most general display format for real numbers, real numbers you display in this format are very difficult to read. Therefore, the HLA *str* module also provides a set of functions that convert real values using the decimal representation. Although you cannot (practically) use these decimal output routines for all real values, they are applicable to a wide variety of common numbers you will use in your programs.

These functions require four parameters: the real value to convert, the width of the converted value, the number of digit positions to the right of the decimal point, and a padding character. These functions convert their values using the following string format:

s	i	i	i	.	f	f	f	f	f	f
---	---	---	---	---	---	---	---	---	---	---

s is a space for positive values, '-' for negative values  
i represents the integer portion of the mantissa  
ffffff represents the fractional portion of the mantissa

```

procedure str.catr32
(
    dest          :string;
    r             :real32;
    width         :uns32;
    decpts        :uns32;
    pad           :char
);

```

This procedure appends a 32-bit single precision floating point value to the *dest* string. The string consumes exactly *width* characters in *dest*. If the numeric output, using the specified number of positions to the right of the decimal point, is sufficiently small that the string representation would be less than *width* characters, then this procedure uses the value of *pad* as the padding character to fill the output with *width* characters. The number is right-justified within the output field (that is, this function prints the padding characters before the string representation of the number).

HLA high-level calling sequence examples:

```

str.catr32( dest, r32Var, width, decpts, fill );
str.catr32( dest, r32Var, 10, 2, '*' );

// If the real32 value is in an FPU register (ST0):

var
    r32Temp:real32;
.
.
.
fstp( r32Temp );
str.catr32( dest, r32Temp, 12, 2, al );

```

HLA low-level calling sequence examples:

```

push( dest );
push( (type dword r32Var) );
push( width );
push( decpts );
movzx( fill, eax );
push( eax );
call str.catr32;

push( dest );
push( (type dword r32Var) );
push( width );
push( decpts );
pushd( (type dword fill) ); // If no memory fault possible
call str.catr32;

push( dest );
sub( 4, esp );
fstp( (type real32 [esp]) );
pushd( 12 );
sub( 4, esp );
push( eax );
movzx( fill, eax ); // If memory fault were possible
mov( eax, [esp+4] ); // in above code.
pop( eax );
call str.catr32;

```

```

procedure str.catr64
(
    dest          :string;
    r             :real64;
    width         :uns32;
    decpts       :uns32;
    pad          :char
);

```

This procedure appends a string representation of a 64-bit double precision floating point value to the *dest* string. The string consumes exactly *width* characters in the output. If the numeric output, using the specified number of positions to the right of the decimal point, is sufficiently small that the string representation would be less than *width* characters, then this procedure uses the value of *pad* as the padding character to fill the output with *width* characters.



HLA high-level calling sequence examples:

```
str.catr64( dest, r64Var, width, decpts, fill );
str.catr64( dest, r64Var, 10, 2, '*' );

// If the real64 value is in an FPU register (ST0):

var
    r64Temp:real64;
.
.
.
fstp( r64Temp );
str.catr64( dest, r64Temp, 12, 2, al );
```

HLA low-level calling sequence examples:

```
push( dest );
push( (type dword r64Var[4]) );
push( (type dword r64Var) );
push( width );
push( decpts );
movzx( fill, eax );
push( eax );
call str.catr64;

push( dest );
push( (type dword r64Var[4]) );
push( (type dword r64Var) );
push( width );
push( decpts );
pushd( (type dword fill) ); // If no memory fault possible
call str.catr64;

push( dest );
sub( 8, esp );
fstp( (type real64 [esp]) );
pushd( 12 );
sub( 4, esp );
push( eax );
movzx( fill, eax ); // If memory fault were possible
mov( eax, [esp+4] ); // in above code.
pop( eax );
call str.catr64;
```

```
procedure str.catr80
(
    dest      :string;
    r        :real80;
    width    :uns32;
    decpts   :uns32;
```

```

    pad      :char
);

```

This procedure appends the string form of an 80-bit extended precision floating point value to the *dest* string. The string consumes exactly *width* characters in the output. If the numeric output, using the specified number of positions to the right of the decimal point, is sufficiently small that the string representation would be less than *width* characters, then this procedure uses the value of *pad* as the padding character to fill the output with *width* characters.

HLA high-level calling sequence examples:

```

str.catr80( dest, r80Var, width, decpts, fill );
str.catr80( dest, r80Var, 10, 2, '*' );

// If the real80 value is in an FPU register (ST0):

var
    r80Temp:real80;
.
.
.
fstp( r80Temp );
str.catr80( dest, r80Temp, 12, 2, al );

```

HLA low-level calling sequence examples:

```

push( dest );
pushw( 0 );
push( (type word r80Var[8]) );
push( (type dword r80Var[4]) );
push( (type dword r80Var) );
push( width );
push( decpts );
movzx( fill, eax );
push( eax );
call str.catr80;

push( dest );
pushw( 0 );
push( (type word r80Var[8]) );
push( (type dword r80Var[4]) );
push( (type dword r80Var) );
push( width );
push( decpts );
pushd( (type dword fill) ); // If no memory fault possible
call str.catr80;

push( dest );
sub( 12, esp );
fstp( (type real80 [esp]) );
pushd( 12 );
sub( 4, esp );
push( eax );
movzx( fill, eax ); // If memory fault were possible
mov( eax, [esp+4] ); // in above code.
pop( eax );
call str.catr80;

```

## 33.16 Generic String Format Output Routine

```
#macro str.put( list_of_items );
```

*str.put* is a macro that automatically invokes an appropriate *str.catXXX* output routine based on the type of the parameter(s) you pass it. This is a very convenient output routine and is probably the *str.catXXX* output call you will use most often in your programs. Keep in mind that this macro is not a single function call; instead, HLA translates this macro into a sequence of calls to procedures like *str.catu32*, *str.cats*, etc.

*str.put* is a macro that provides a flexible syntax for outputting data to a string. This macro allows a variable number of parameters. For each parameter present in the list, *str.put* will call the appropriate routine to emit that data, according to the type of the parameter. Parameters may be constants, registers, or memory locations. You must separate each macro parameter with a comma.

Here is an example of a typical invocation of *str.put*:

```
str.put( dest, "I=", i, " j=", j, nl );
```

The above is roughly equivalent to

```
str.cpy( "", dest );
str.cats( dest, "I=" );
str.catu32( dest, i );
str.cats( dest, " j=" );
str.catu32( dest, j );
str.cats( dest, nl );
```

This assumes, of course, that *i* and *j* are int32 variables.

The *str.put* macro also lets you specify the minimum field width for each parameter you specify. To print a value using a minimum field width, follow the object you wish to print with a colon and the value of the minimum field width. The previous example, using field widths, could look like the following:

```
str.put( dest, "I=", i:2, " j=", j:5, nl );
```

Although this example used the literal decimal constants two and five for the field widths, keep in mind that register values and memory value (integers, anyway) are perfectly legal here.

For floating point numbers you wish to display in decimal form, you can specify both the minimum field width and the number of digits to print to the right of the decimal point by using the following syntax:

```
str.put( "Real value is ", f:10:3, nl );
```

The *str.put* macro can handle all the basic primitive types, including boolean, unsigned (8, 16, 32, 64, 128), signed (8, 16, 32, 64, 128), character, character set, real (32, 64, 80), string, and hexadecimal (byte, word, dword, qword, lword).

There is a known "design flaw" in the *str.put* macro. You cannot use it to print HLA intermediate variables (i.e., non-local VAR objects). The problem is that HLA's syntax for non-local accesses takes the form "reg32:varname" and *str.put* cannot determine if you want to print *reg32* using *varname* print positions versus simply printing the non-local *varname* object. If you want to display non-local variables you must copy the non-local object into a register, a static variable, or a local variable prior to using *str.put* to operate on it. Of course, there is no problem using the other *str.catXXXX* functions to display non-local VAR objects, so you can use those as well.



## 34 High-Level Language Module (hll.hhf)

The hll.hhf library module adds a switch/case/default/endswitch statement that is similar to the Pascal case statement and the C/C++ switch statement.

### 34.1 The HLL Module

To use the high-level language functions in your application, you will need to include one of the following statements at the beginning of your HLA application:

```
#include( "hll.hhf" )
or
#include( "stdlib.hhf" )
```

### 34.2 The switch/case/default/endswitch Macro

```
#macro switch( reg32 );
#keyword case( const_list );
#keyword default
#terminator endswitch
```

A commonly used high level language statement missing from HLA's basic set is the the C/C++ switch statement (the case statement in most other languages). The SWITCH/CASE/DEFAULT/ENDSWITCH macro set in the hll.hhf header file provides this missing HLL statement.

The HLL module's switch statement actually provides two different user-selectable syntaxes. The first is a Pascal-like syntax. It takes the following form:

```
switch( reg32 )

    case( constant_list )
        <<body>>

    << additional, optional cases >>

    default          // This section is optional too!
        << body >>

endswitch;
```

As you might expect, the reg<sub>32</sub> parameter has to be an 80x86 32-bit general purpose register. The *constant\_list* operand has to be a sequence of one or more positive ordinal constants. There must be at least one *case* present in the statement (default does not count as a case) and there may be a maximum of 1,024 cases in the *switch* statement. Furthermore, the range between the largest and smallest values for all the cases must be less than or equal to 1,024. Note that, unlike C/C++, you do not end each case with a *break* statement; nor does control fall through from one case to the next. Here is a simple example of a Pascal-like *switch* statement:

```
switch( ebx )

    case( 1 )
        stdout.put( "case 1 encountered" nl );

    case( 3 )
        stdout.put( "case 3 encountered" nl );

    case( 10 )
        stdout.put( "case 10 encountered" nl );
```

```

    case( 15, 20, 25 )
        stdout.put( "case 15, 20, or 25 encountered" nl );

    default
        stdout.put( "Some other case was encountered" nl );

endswitch;

```

Although C/C++ semantics for a switch statement are stylistically inferior to Pascal, some people might prefer a C/C++ version of the switch statement. The HLL switch statement uses a special predefined boolean VAL constant, *hll.cswitch*, that lets you choose C/C++ semantics. By default, the *hll.cswitch* constant is set to false. By placing the statement "*?hll.cswitch:=true;*" before a *switch* statement, you can instruct the *switch* macro to use C/C++ semantics rather than Pascal semantics. The difference between the two is that for C/C++ semantics you must end each case with an explicit *break* statement. The Pascal version is preferable since it is slightly more efficient and a bit more readable.

By default, the *switch* macro uses a quicksort algorithm built into HLA's *@sort* compile-time function to sort the cases when building the jump table that the *switch* statement compiles into. For the vast majority of *switch* statements you'll write, this is a good choice. However, if you create a really large *switch* statement and the cases you supply are already sorted in ascending order (or mostly sorted), a bubble sort will actually outperform the quick sort algorithm. In this (very) special case, you can improve the compilation (not run-time) performance of the *switch* macro by adding the following statement immediately after the switch statement:

```

switch( eax )
    ?hll.usebubblesort := true;

    <lots of pre-sorted cases>

endswitch;

```

Note that this trick speeds up compilation only if the cases are already sorted in ascending order. If they are not sorted in ascending order, then the bubblesort algorithm is *much* slower than the quicksort algorithm and you shouldn't use it.

## 35 Tables Module (tables.hhf)

The HLA Tables module provides support for associative lookup tables. You can view a table as an array of objects that uses a string index rather than an integer index. The HLA table routines use a hash table to rapidly look up the specified string in the table and return a pointer to the specified element in the table.

Note: Because of their high-level nature, this document only provides high-level calling sequences for the table management procedures. Low-level calls are possible, but are generally so painful that they aren't worth making. If you are dead set on making low-level calls to *table* class methods and procedures, please consult the HLA documentation for directions on how this is done.

**A Note About Thread Safety:** The HLA standard library table module does not attempt to synchronize thread access to the table data structures. If you are going to be manipulating tables from multiple threads, it is your responsibility to ensure that the threads use properly synchronized access to this resource. This issue may be addressed in a future version of the standard library, for now it is your responsibility to ensure correct operation in a multi-threaded environment.

### 35.1 The Tables Module

To use the table class and methods in your application, you will need to include one of the following statements at the beginning of your HLA application:

```
#include( "tables.hhf" )
or
#include( "stdlib.hhf" )
```

### 35.2 The Table Class

The HLA stdlib tables module consists of two classes: the *table* class and the *node* class. To create a table object, you first create a new *node* type by overloading the *node* class and then you can use the *table* class' methods to manipulate a table of these nodes.

To begin with, each element (called a *node*) in an HLA table uses the following structure:

```
tableNode_t:
    record

        link:      pointer to tableNode_t;
        Value:      dword;
        id:         string;

    endrecord;
```

The *link* field is for internal use by the table management routines; you should never modify its value.

The *id* field points at the string that indexes the current node in the table. You may use the value of this string, but you must never modify the characters in the string. The hashing function utilized by the table management code will not be able to locate this string if you change the characters in the string after entering the string into the table.

The *Value* field is reserved for your own purposes. Other than initializing this field to zero when they create a table entry, the table management routines ignore this field. If you wish to associate data with an entry in the table you can either store the data directly in this field (if the data is four bytes or less), or you can allocate storage for the data outside the table entry and store a pointer to the data in this field (remember, pointers are four-byte objects).

The *table\_t* data type is a class that provides the following methods and procedures:

```
procedure table_t.create( HashSize:uns32 );
method table_t.destroy( FreeValue:procedure );
method table_t.getNode( id:string );
method table_t.lookup( id:string );
iterator table_t.item();
```

Like most HLA classes, the `table_t` class provides a constructor named `table_t.create` and a destructor named `table_t.destroy`. The class also provides two additional methods and an iterator, `table_t.getNode`, `table_t.lookup`, and `table_t.item`. Although there doesn't appear to be much to this class, these few routines provide considerable power.

```
procedure table_t.create( HashSize:uns32 );
```

The create procedure, being a static procedure constructor, is typically called in one of two fashions:

When dynamically allocating a `table_t` object on the heap and storing the pointer away into a variable whose type is "pointer to `table_t`":

```
table_t.create( some_constant );
mov( esi, PtrToTableVar );
```

When you've got a var or static variable object (named `table_var_name` in this example), you can use code like the following to construct the `table_t` object:

```
table_var_name.create( some_constant );
```

The `table_t` constructor requires a single `uns32` parameter. This value should be approximately the number of entries (elements) you expect to insert into the table. This value does not have to be exact. Anytime you want to add a new element to the table, you may do so; there are no limitations (other than available memory) on the number of elements in a table. However, this parameter value is a hint to the table management routines so it can allocate a hash table of an appropriate size so that (1) access to the table elements is fast, and (2) the hash table doesn't waste an inordinate amount of space. If the hint value you supply is too small, the table lookup routines will still function properly, but they will run a little slower. If the hint value you supply is too large, the table management routines will waste some memory.

HLA high-level calling sequence example:

```
table_t.create( 128 );
mov( esi, someTableVarName );

SomeStaticTableVarName.create( 256 );
```

```
method table_t.destroy( FreeValue:procedure );
```

The `table_t.destroy` method frees up the data in the table. This routine deallocates the storage associated with the hash table, it deallocates the storage associated with each node in the table, and it deallocates the storage associated with each string (the `id` field in record `tableNode_t` above) in the table. Unfortunately, the `table_t.destroy` method doesn't know anything at all about the `Value` field of each node. If this is just some simple data, then the destructor probably doesn't need to do anything with the `Value` field. On the other hand, if `Value` is a pointer to some other data that was dynamically allocated, `destroy` should probably deallocate the storage associated with the `Value` field. Unfortunately, `destroy` has no apriori knowledge about the `Value` field, so it cannot determine if (or how) it should deallocate storage associated with `Value`.

To resolve the problem above, `table_t.destroy` calls a user-defined function that is responsible for cleaning up the data associated with `Value`. You will notice that `table_t.destroy` has a single parameter: `FreeValue`. This parameter must be the address of a procedure whose job is to handle the destruction of the `Value` field. If no clean up is necessary, you must still provide the address of some routine. That routine should simply return without further activity.

Upon entry into the `FreeValue` procedure, the `EBX` register contains a pointer to the current `tableNode_t` record being deallocated. At this point, none of the other fields have been modified; in particular, the `id` field is still pointing at the string associated with the node. The `FreeValue` procedure may access the `id` field, but it must not deallocate the storage associated with this string. The `table_t.destroy` method takes care of that after `FreeValue` returns.

The "tabledemo.hla" file accompanying the HLA release gives another example of how you could use the `FreeValue` procedure. This code doesn't deallocate any storage in this procedure (named `PrintIt` in this file), instead, it uses this call to dump the data associated with the node to the display when the table is freed.



HLA high-level calling sequence example:

```
// Dummy free routine, nothing to do:
```

```
procedure myFree; @noframe;
begin myFree;
    ret();
end myFree;
```

```
.
.
.
```

```
table_t.create( 128 );
mov( esi, myTable );
```

```
.
.
.
```

```
myTable.destroy( &myFree );
```

**method table\_t.lookup( id:string ); @returns( "eax" );**

The *table\_t.lookup* method locates a node in the table. The string parameter specifies which node to find in the table. On return, EAX contains the address of the corresponding *tableNode\_t* record in the table, if the specified node is present (that is, the node's *id* field matches the string passed to *table\_t.lookup*). If the node is not present in the table, then the *table\_t.lookup* method returns NULL (zero) in the EAX register.

HLA high-level calling sequence example:

```
table_t.create( 16 );
mov( eax, tableVar );
```

```
.
.
.
```

```
tableVar.lookup( "someString" );
if( eax <> NULL ) then
```

```
    stdout.put( "Found 'someString' in the table" nl );
```

```
else
```

```
    stdout.put( "Did not find 'someString' in the table" nl );
```

```
endif;
```

**method table\_t.getNode( id:string ); @returns( "eax" );**

The *table\_t.getNode* method serves two purposes. First of all, it looks up the specified string value in the table. If it finds a node in the table corresponding to the string parameter, it returns a pointer to that (*table\_t.tableNode\_t*) node in the EAX register. If it does not find the string in the table, then it creates a new node and inserts the string into that new node; it also initializes the *Value* field to zero in this case. Note that *table\_t.getNode* makes a copy (using *str.a\_cpy*) of the string, it does not store the string pointer you pass directly into the *id* field. Upon return, EAX will point at the new node. Note that whether or not an existing node is present in the table, *table\_t.getNode* will always return a pointer to the node associated with the specified string. It either returns a pointer to a pre-existing node or it returns the pointer to the new node.

If you want to insert a new node into the table and fail if the node already exists, you will need to first call *table\_t.lookup* to see if the node previously exists (and fail if it does). You may then call *table\_t.getNode* to insert a new node into the table. While it would be easy to add this functionality to the *table\_t* class, it would be rarely used and probably isn't needed.

HLA high-level calling sequence example:

```
table_t.create( 16 );
mov( eax, tableVar );
tableVar.getNode( "someString" );// Insert "someString"
.
.
.
tableVar.getNode( "someString" );// Retrieve ptr to "someString"
```

**method table\_t.getNode( id:string ); @returns( "eax" );**

The *table\_t.item* iterator yields each node in the table during the execution of the corresponding foreach loop. Note that *table\_t.item* does not yield the nodes in any particular (discernable) order. However, it will yield each item in the list exactly once. The iterator returns with EAX pointing at a *table\_t.tableNode\_t* object.

HLA high-level calling sequence example:

```
table_t.create( 16 );
mov( eax, tableVar );
.
.
.
foreach tableVar.item() do

    // Process node pointed at by EAX.

endfor;
```

## 36 Threads Module (threads.hhf)

The HLA Threads module provides a set of routines that let you create, control, and synchronize multiple threads in an application.

**A Note About Thread Safety:** While the routines in the thread library are (mostly) thread safe, keep in mind that you must be linking in a thread-safe version of the HLA Standard Library if you expect calls to other functions to operate in a thread-safe manner.

### 36.1 Threads Module

To use the thread functions in your application, you will need to include the following statement at the beginning of your HLA application:

```
#include( "threads.hhf" )
```

Note that the "stdlib.hhf" header file does not automatically include the threads.hhf header file. This is because simply including the "threads.hhf" header file may force the inclusion of considerable code, even if you do not call any functions in the thread library. Therefore, you must explicitly include the threads.hhf header file if you want to call thread functions in the HLA Standard Library.

If you are using thread functions in your application, you must use the "-thread" command-line parameter to force HLA to link in the thread-safe version of the HLA Standard Library. Failure to do so will probably cause the linker to fail; even if you manage to get the program to link properly, you'll link in non-thread-safe versions of the HLA Standard Library functions and this will probably cause your program to fail or otherwise misbehave.

The functions in the threads module are broken down into six types: thread creation, thread identification, thread local storage, events, critical section maintenance, and semaphore maintenance. The following sections will describe each of these categories.

### 36.2 Thread Creation

```
procedure thread.create( func:threadFunc_t; parm:dword; stackSize:dword );
@returns( "eax" );
```

This function creates a new thread. The *func* parameter is the address of an HLA procedure where the thread will begin execution. This function has the following prototype:

```
type
  threadFunc_t:procedure( parm:dword );
```

That is, the thread function must have a single double word parameter.

The *parm* argument (to *thread.create*) is passed along to the thread function specified by the *func* argument. This can be any 32-bit value you want. Often, this argument is a pointer to some global data you're supplying to the thread. Keep in mind, however, that the thread may not begin executing before *thread.create* returns to its caller. You must ensure that any data whose address you pass in the *parm* argument remains valid as long as the new thread requires that data. In particular, do not pass the address of some local variables allocated on the stack that might go away when the procedure that calls *thread.create* returns to its caller.

The *stackSize* parameter specifies the number of bytes of storage that will be allocated for the stack when the thread is created by the operating system. This value should be a multiple of 4,096 bytes. If you specify zero, the system will assign a default value (the default value is OS dependent). Unless your thread requires very little thread storage, you should always supply a value for the *stackSize* parameter. Note that the HLA Standard Library will allocate storage for its own thread local variables on the stack created for the thread. Therefore, you should allocate an additional 4,096 bytes above and beyond your own needs to provide sufficient storage for the stdlib thread local objects.

The thread creation function returns a thread identifier in the EAX register. This information is useful when you have multiple threads executing the same code (that is, the same thread function) and you need to pass information to a specific thread. The individual threads can determine their own thread ID and use that to determine whether a packet of information is intended for them (see the discussion of the *thread.getCurrentThreadHandle* function for details).

A thread terminates execution by returning from the thread function. If you want to prematurely terminate a thread, then that thread must jump to the end of the procedure and return from it (or clean up the stack and execute a RET instruction). Note that the HLA Standard Library does not allow one thread of execution to terminate another thread (including the parent thread that started the thread in the first place). If you want to terminate one thread under the control of another thread, then you must pass some message to that thread and tell it to terminate itself.

Each thread of execution has its own exception handling system. Exceptions that the system or program raises in one thread must be handled by that thread. There is no way to pass exceptions on to a different thread (including the parent thread). If an exception occurs in a thread and you do not provide an exception handler (via a **try..endtry** statement), then the system will abort execution of the whole application.

The following examples assume the presence of the following thread function:

```
procedure myThreadFunc( theParm:dword );
var
    myThreadID:dword;
begin myThreadFunc;

    try

        thread.getCurrentThreadHandle();
        mov( eax, myThreadID );

        // Code that does something for this thread

    anyexception

        stdout.put
        (
            "Thread $",
            myThreadID,
            " terminated with exception $",
            eax,
            nl
        );

    endtry;

end myThreadFunc;
```

HLA high-level calling sequence examples:

```
thread.create( &myThreadFunc, 0, 0 ); // Default stack size
mov( eax, childThreadID );
```

HLA low-level calling sequence examples:

```
pushd( &myThreadFunc );
pushd( 0 );                // parm value = 0
pushd( 0 );                // Default stack size
call thread.create;
mov( eax, childThreadID );
```

## 36.3 Thread Identification

```
procedure thread.getCurrentThreadHandle; @returns( "eax" );
```

This function returns a unique thread identifier value in the EAX register for the current thread. This value matches the value that *thread.create* returns (for the child thread, assuming that the child thread is the one calling *thread.getCurrentThreadHandle*). You should not assume that the value that this function returns is the same as the thread ID used by the underlying OS.

HLA high-level calling sequence examples:

```
thread.getCurrentThreadHandle();
mov( eax, myThreadID );
```

HLA low-level calling sequence examples:

```
call thread.getCurrentThreadHandle;
mov( eax, myThreadID );
```

## 36.4 Thread Local Storage

Sometimes various procedures in a thread need to communicate data amongst themselves using static storage. Unfortunately, you must be very careful about using static objects in a threaded application. In general, using static objects renders the application thread unsafe (that is, causes the program to fail when two or more threads attempt to use the same static object).

A good example of this problem occurs in the HLA Standard Library. Consider the *underscores* variable that determines if a hexadecimal numeric conversion routine should emit underscores between groups of four hexadecimal digits. This is a global value that all of the hexadecimal conversion routines in the Standard Library reference. Now suppose you have two threads that call these conversion routines. If one thread turns underscore output on and the other thread turns it off in the middle of a conversion in the other thread, the conversion will be incorrect. The solution is to give each thread its own copy of the *underscores* variable. When one thread turns this feature on, it does not affect the conversions in any of the other threads. Unfortunately, giving each thread its own copy of the *underscores* variable is a lot more difficult than it sounds. You cannot use static objects for this purpose – the different threads will all access the same static objects. You cannot allocate the *underscores* variable on the stack in your thread, the various procedures won't be able to access that object without knowing its exact address in memory.

The solution is *thread local storage*. With thread-local storage (TLS) you request (once) a TLS context handle from the operating system. With this handle you can store a 32-bit thread-local value and retrieve that 32-bit thread-local value. Normally, you don't store the actual thread-local data via this handle, instead you store a pointer to a data structure containing all the thread-local data. This structure can be allocated on the heap or you can allocate it in the VAR section of your main thread. Because you can retrieve the address of this data structure via TLS calls, various functions can figure out the address of the global objects they're interested in via some offset from the address of the structure.

Note that you only need to obtain a single TLS context handle – you do not obtain a separate handle for each thread in your application. Your main thread should obtain this context handle before spawning any other child threads. You should store the value of this context handle in a global, static, object that all threads can access. You will never modify this object directly; instead, you will pass the address of the object to the *thread.createTLS* function and then your threads will only read this value from that point forward (which is a safe use of a global, static, object in a threaded application).

```
procedure thread.createTLS( var context:dword );
```

This function asks the operating system to create a thread-local storage context handle. You pass the address of the context variable to this function as the single parameter and the *thread.createTLS* function will automatically (and atomically) update that variable. Note that you should create (initialize) the context handle before creating any threads that might use it. Otherwise you might create a race condition where the main thread stops, a child thread runs, and the child thread attempts to use the context handle before the main thread initializes it.

HLA high-level calling sequence examples:

```
thread.createTLS( contextHandle );
```

HLA low-level calling sequence examples:

```
lea( eax, contextHandle );
push( eax );
call thread.createTLS;
```

```
procedure thread.setTLS( context:dword; valueToSet:dword );
```

The *thread.setTLS* function stores the value found in the *valueToSet* parameter into a thread-local double word value specified by the *context* parameter. Generally, you will pass the address of some block of memory (some data structure) as the *valueToSet* parameter. If you allocate that block of memory on the heap (e.g., via *mem.alloc*) or on your local stack, any function in the current thread can access that data structure later by calling the *thread.getTLS* function.

Although you can use *thread.setTLS* to set a single variable value (rather than setting the address of some data structure allocated for the current thread), you can only access 32 bits of data this way. As a result, most programmers store an address to some data structure via *thread.setTLS* rather than directly storing data.

If you use *thread.setTLS* in the normal manner, by allocating some storage and storing an address away, you should allocate the storage at the very beginning of your thread's main function. If this is a fixed-size data structure, you can allocate it as part of your local variables (in the VAR section) and simply take the address of the structure (e.g., with the LEA instruction) and pass that as the *valueToSet* argument. If the data structure is variable in size, then *mem.alloc* is probably a good choice for allocating the data structure.

HLA high-level calling sequence examples:

```
lea( eax, myLocalData );
thread.setTLS( contextHandle, eax );
```

HLA low-level calling sequence examples:

```
push( contextHandle );
lea( eax, myLocalData );
push( eax );
call thread.setTLS;
```

```
procedure thread.getTLS( context:dword ); @returns( "eax" );
```

The *thread.getTLS* function returns the value associated with the thread context handle passed as the single argument. This value is set via an earlier call to the *thread.setTLS* function. See the discussion of *thread.setTLS* for more details.

HLA high-level calling sequence examples (assuming the thread local storage value has been set to point at an object of type "someDataType" by a previous call to *thread.setTLS*):

```
thread.getTLS( contextHandle );
mov( (type someDataType [eax]).someField, ecx );// Retrieve data
```

HLA low-level calling sequence examples:

```
push( contextHandle );
call thread.getTLS;
mov( (type someDataType [eax]).someField, ecx );// Retrieve data
```

## 36.5 Events

The HLA Standard Library threads module provides a thread synchronization system known as events. An event is something that a thread can wait upon until a different thread signals (or sets) the event. An HLA stdlib event is an object that various threads can use to coordinate events in the program.

To use events, a program must first create an event object via a call to *thread.createEvent*. This initializes the event and puts it in the non-signaled/non-set state. The *thread.createEvent* returns an OS event handle that you will save to allow various threads to work with that event object. When you are done using a thread, you can tell the OS to reclaim the resources used by the event object.

When you create an event, it is initialized in an unsignaled state. Whenever some thread waits on an event (by calling *thread.waitForEvent*), that thread will block (suspend) until some other thread signals the event by calling *thread.setEvent*. If two or more threads are waiting for an event, only one thread will be placed in the executable state when an event is signaled. Additional calls to *thread.setEvent* will be necessary to resume any other threads waiting for the event.

Whenever a thread waiting on an event resumes execution after that event has been signaled, the system automatically sets the event to the unsignaled state. There is no explicit call you can make to "unsignal" or unset an event.

```
procedure thread.createEvent; @returns( "eax" );
```

The *thread.createEvent* function creates an event object and returns a handle to that object in the EAX register. All events you use must be initialized via a call to *thread.createEvent*. Note that event initialization consumes resources internal to the OS' thread library. You should call *thread.deleteEvent* to reclaim those resources when you are done using the event object you've created via *thread.createEvent*.

HLA high-level calling sequence examples:

```
thread.createEvent();
mov( eax, eventHandle );
```

HLA low-level calling sequence examples:

```
call thread.createEvent;
mov( eax, eventHandle );
```

```
procedure thread.deleteEvent( event:dword );
```

The *thread.deleteEvent* function reclaims all OS/library resources in use by an event object. You should call this function after you are done using an event object. You must not call this function on an event handle if any threads are waiting on the event. Of course, you must not continue to use the event handle after deleting the event.

HLA high-level calling sequence examples:

```
thread.deleteEvent( eventHandle );
```

HLA low-level calling sequence examples:

```
push( eventHandle );
call thread.deleteEvent;
```

```
procedure thread.setEvent( event:dword );
```

The *thread.setEvent* function signals the occurrence of an event. If some thread is waiting for this event to occur, this call will resume the execution of that thread. If more than one thread is waiting on the event, then only one thread will be unblocked and allowed to execute. In order to release all threads waiting on an event, you must call *thread.setEvent* once for each blocked thread. It is the application's responsibility to count the number

of threads waiting on an event and call *thread.setEvent* an appropriate number of times. Calling *thread.setEvent* multiple times without having any threads waiting on the event between signaling the event is undefined. Some OS thread APIs might count the number of calls and release that many threads that (ultimately) wait for the event. Other OS thread packages might ignore multiple requests and release only one thread that waits on the event. Still other OSes may completely ignore the call to *thread.setEvent* if there are no threads waiting on that particular event.

HLA high-level calling sequence example:

```
thread.setEvent( eventHandle );
```

HLA low-level calling sequence examples:

```
push( eventHandle );
call thread.setEvent;
```

```
procedure thread.waitForEvent( event:dword );
```

The *thread.waitForEvent* function blocks (suspends the execution of) the current thread until some other thread signals (sets) the event with a call to *thread.setEvent*. Note that the results are undefined if a call is made to *thread.setEvent* prior to some other thread waiting on that event. The system may choose to ignore the earlier call to *thread.setEvent* or it may immediately resume the thread and return from a call to *thread.setEvent*. The exact semantics are OS-dependent.

HLA high-level calling sequence example:

```
thread.waitForEvent( contextHandle );
```

HLA low-level calling sequence examples:

```
push( contextHandle );
call thread.waitForEvent;
```

## 36.6 Critical Sections

Critical sections are synchronization objects that ensure that only one thread at a time executes a protected section of code (or accesses some data structure). A thread *enters* and *leaves* a critical section. While one thread is holding a critical section lock, an attempt by some other thread to enter that same critical section causes the second thread to block until the first thread leaves the critical section.

As for all synchronization objects the HLA stdlib supports, an application must first create a critical section object to obtain a handle to be used when entering and leaving critical sections. Because the creation of a critical section allocates some system resources, the application should delete the critical section object when it is done using it. The *thread.createCriticalSection* and *thread.deleteCriticalSection* functions handle these chores.

Once you've created a critical section object via *thread.createCriticalSection*, you can synchronize threads using the *thread.enterCriticalSection* and *thread.leaveCriticalSection* functions. To protect a sequence of instructions (that, perhaps, operate on a protected data structure) you would call the *thread.enterCriticalSection* (passing it the handle of a critical section object you've created) to lock the use of that particular critical section object. When you are done executing the protected code, you call the *thread.leaveCriticalSection* function to release the lock. If any other thread attempts to enter the same critical section (by calling *thread.enterCriticalSection* and passing in the same critical section handle), then that second thread will block until the first thread calls *thread.leaveCriticalSection* and releases the lock.

If one thread is holding a critical section lock and two or more additional threads attempt to enter the same critical section, all those new threads will block. When the thread holding the lock calls *thread.leaveCriticalSection*, only one of the waiting threads will be activated to resume execution. Note that the



order of thread activation is not defined and you should not assume that the first blocked thread will be the first to be released. The only guarantee is that exactly one thread will be released.

**procedure thread.createCriticalSection; @returns( "eax" );**

The *thread.createCriticalSection* function creates a critical section object and returns a handle to that object in the EAX register. All critical sections you use must be initialized via a call to *thread.createCriticalSection*. Note that critical section initialization consumes resources internal to the OS' thread library. You should call *thread.deleteCriticalSection* to reclaim those resources when you are done using the critical section object you've created via *thread.createCriticalSection*.

HLA high-level calling sequence examples:

```
thread.createCriticalSection();
mov( eax, csHandle );
```

HLA low-level calling sequence examples:

```
call thread.createCriticalSection;
mov( eax, eventHandle );
```

**procedure thread.deleteCriticalSection( csHandle:dword );**

The *thread.deleteCriticalSection* function reclaims all OS/library resources in use by a critical section object. You should call this function after you are done using a critical section object. You must not call this function on a critical section handle if any threads are executing in the critical section. Of course, you must not continue to use the critical section handle after deleting the critical section.

HLA high-level calling sequence examples:

```
thread.deleteCriticalSection( csHandle );
```

HLA low-level calling sequence examples:

```
push( csHandle );
call thread.deleteCriticalSection;
```

**procedure thread.enterCriticalSection( csHandle:dword );**

The *thread.enterCriticalSection* function first checks to see if some other thread has already entered the critical section specified by the *csHandle* parameter. If this is the case, then the current thread (that is calling *thread.enterCriticalSection*) blocks until the first thread releases the critical section handle (that is, it leaves the critical section) by calling *thread.leaveCriticalSection*. If no other thread currently holds the critical section lock, or if the current thread resumes execution because some other thread releases the lock (by calling *thread.leaveCriticalSection*), then the current thread obtains the lock and execution resumes with the first instruction after the call to *thread.enterCriticalSection*.

HLA high-level calling sequence examples:

```
thread.enterCriticalSection( csHandle );
```

HLA low-level calling sequence examples:

```
push( csHandle );
call thread.enterCriticalSection;
```

```
procedure thread.leaveCriticalSection( csHandle:dword );
```

The *thread.leaveCriticalSection* function releases the critical section lock specified by the *csHandle* parameter. This allows any other thread that is waiting on the critical section to resume execution (and obtain the critical section lock).

HLA high-level calling sequence examples:

```
thread.leaveCriticalSection( csHandle );
```

HLA low-level calling sequence examples:

```
push( csHandle );
call thread.leaveCriticalSection;
```

## 36.7 Semaphores

Semaphores are the generic process/thread synchronization mechanism. Semaphores provide two main extensions over other synchronization objects provide by the HLA stdlib:

- HLA stdlib semaphores allow synchronization of processes (e.g., different applications) as well as threads.
- HLA stdlib semaphores are counting semaphores, allowing
- *n* processes access to some protected resource (where *n* is some value you specify when creating the semaphore).

Like the other HLA stdlib synchronization objects, you must create a semaphore object before using it and you must delete a semaphore object when you are done using it. The *thread.createSemaphore* and *thread.deleteSemaphore* functions handle these chores. When creating the semaphore, you specify the number of resources the semaphore will protect (that is, the number of threads that can concurrently hold the semaphore and run without blocking). You also specify a (system-wide) semaphore name when creating the semaphore; the operating system uses this name to connect semaphore objects in different processes to the same system-wide semaphore object (that is, if two processes specify the same name for the semaphore object, then those two processes will access the exact same semaphore).

To obtain a resource held be a semaphore, you will call the *thread.waitSemaphore* function. This function blocks if there are no resources available, it will decrement the resource count and return if resources are available. To release a semaphore resource that a thread is holding, the thread executes the *thread.releaseSemaphore* function.

```
procedure thread.createSemaphore( maxCnt:dword; semName:string );
    @returns( "eax" );
```

The *thread.createSemaphore* function creates a semaphore object and returns a handle to that object in the EAX register. All future access to that semaphore will be via the handle value that *thread.createSemaphore* returns in the EAX register.

The *maxCnt* parameter specifies the number of threads (or processes) that may concurrently hold the semaphore before the operating system blocks any further semaphore requests. If *maxCnt* is one, then the semaphore behaves in a manner similar to a critical section insofar as it only allows access to the lock by one thread (or process) at a time. This is known as a binary semaphore.

The *semName* string parameter provides a system-wide name for the semaphore. This string should correspond to an existing filename in the system for best results (create an empty file that has the semaphore's name if you want to create a semaphore using a name other than that of some existing file). If two different calls to the *thread.createCriticalSection* function specify the same semaphore name, then the handle value that this function returns will refer to the same semaphore object.

If multiple calls to *thread.createCriticalSection* specify the same string for *semName* but specify different values for *maxCnt*, then the result is undefined. The system may use the last value specified, the first value specified, or any other value it pleases.

All semaphore objects you use must be initialized via a call to *thread.createSemaphore*. Note that semaphore initialization consumes resources internal to the OS' thread library. You must call *thread.deleteSemaphore* to reclaim those resources when you are done using the critical section object you've created via *thread.createSemaphore*.

**Warning:** because of a known issue in the UNIX SYSV semaphore interface (that the HLA stdlib used under Linux, Mac OSX, and FreeBSD), there is a brief time period during semaphore creation between the creation of a semaphore object and the setting of the resource count where the system could be interrupted. If another process executes between these two points and specifies a different semaphore count, the results could be unexpected. This is yet another reason why you want to specify the same count value when requesting multiple semaphore handles using the same semaphore name. For this same reason, you should try to allocate all semaphore handles in your main program, before spawning any additional threads (though this won't help much if multiple processes or applications are creating the same semaphore).

HLA high-level calling sequence examples:

```
thread.createCriticalSection();
mov( eax, csHandle );
```

HLA low-level calling sequence examples:

```
call thread.createCriticalSection;
mov( eax, eventHandle );
```

**procedure thread.deleteSemaphore( semHandle:dword );**

The *thread.deleteSemaphore* function will (possibly) delete system resources in use by a semaphore. The exact operation of this command is system dependent. Under Windows, it will decrement a reference counter and if the current process is the last process using the semaphore, this call will free up all system resources used by the semaphore.

As this document was being written, this function is a no-operation under \*NIX operating systems. For those operating systems you will need to manually delete the semaphore object using the `ipcrm` command (use `ipcs` to list the semaphores currently in use by the system). Even if you are using a \*NIX operating system, you should call *thread.deleteSemaphore* because the semantics of this function might change in future versions of the HLA stdlib.

HLA high-level calling sequence examples:

```
thread.deleteSemaphore( semHandle:dword );
```

HLA low-level calling sequence examples:

```
push( semHandle );
call thread.deleteSemaphore;
```

**procedure thread.waitSemaphore( semHandle:dword );**

The *thread.waitSemaphore* function decrements the semaphore resource count (initialized to the value of the *maxCnt* parameter by the *thread.createSemaphore* function). If the result is less than or equal to zero, then this function returns and execution continues. If the result is negative, then this function blocks the current thread until some process releases the semaphore. When the process is done using the resource protected by the semaphore, it should release that resource via a call to the *thread.releaseSemaphore* function.

If a thread wishes to grab multiple resources protected by a semaphore, it can make multiple calls to *thread.waitSemaphore*. It must make the corresponding number of calls to *thread.releaseSemaphore* to release it of those locks it allocates. Obviously, a thread should not call *thread.waitSemaphore* more than *maxCnt* times (*maxCnt* being the parameter value passed to *thread.createSemaphore*) otherwise deadlock will occur.

HLA high-level calling sequence examples:

```
thread.waitSemaphore( semHandle:dword );
```

HLA low-level calling sequence examples:

```
push( semHandle );  
call thread.waitSemaphore;
```

**procedure thread.releaseSemaphore( semHandle:dword );**

The *thread.releaseSemaphore* function increments the internal resource count associated with the semaphore specified by *semHandle*. If there are any threads or processes blocked and waiting on that semaphore, then exactly one of those threads will be placed in an active (runnable) state and allowed to continue execution. Each process/thread should have a corresponding *thread.releaseSemaphore* call for each *thread.waitSemaphore* call it makes.

HLA high-level calling sequence examples:

```
thread.releaseSemaphore( semHandle:dword );
```

HLA low-level calling sequence examples:

```
push( semHandle );  
call thread.releaseSemaphore;
```

## 37 Time Functions (datetime.hhf)

HLA contains a set of procedures and functions that simplify *correct* time calculations. The time module contains functions and other objects that manipulate time in terms of hours, minutes, and seconds. This includes functions that read the current time, perform time arithmetic, do time conversions, and output time values.

There are two sets of time functions available in the standard library: the standard time functions and a set of time classes. This document will describe both sets of time functions.

Note: be sure to read the chapter on "Passing Parameters to Standard Library Routines" (parmpassing.rtf) before reading this chapter.

**A Note About Thread Safety:** The date and time routines maintain a couple of static global variables that track the output format and output separate characters for dates. Currently, these values apply to all threads in a process. You should take care when changing these values in threads. When the process module is added to the standard library, these values will be placed in a per-thread data structure. Until then, you should set the format/separator character before starting any other threads and avoid changing their values once other threads (that might use the date/time library module) begin execution.

**Note about function overloading:** the functions in the date/time module use function overloading in order to allow you to specify the parameter lists in different ways. The macro that handles the overloading generally coerces the possible parameter types into a single object that it passes to the underlying function. The documentation for the specific functions will tell you whether a symbol is a macro or a function. For the most part, this should matter to you unless you are taking the address of a function (which you cannot do with a macro). See the HLA documentation for more details on function overloading via macros.

### 37.1 Time Module

```
#include( "datetime.hhf" )
or
#include( "stdlib.hhf" )
```

### 37.2 Time Data Types

The principal time data structure is the *time.timerec* record:

**time.timerec**

This data structure has the following definition:

```
type
  timerec:
    daterec:
      record
        day      :uns8;
        month    :uns8;
        year     :uns16;
      endrecord;
```

The *time* field allows you to treat the entire object as a single 32-bit value. This is great for comparisons or for passing the *timerec* value around in an opaque fashion.

The standard library uses the *time.timerec* data type to hold valid times in the range 00:00:00 to 23:59:59. Values outside this range are invalid and the standard library will raise an exception if you try to use such values in a *time.timerec* object. Sometimes, however, it is convenient to measure time as a duration rather than as a time of day. The standard library provides the *time.duration* data type for this purpose. The *time.duration* data type is structurally identical to the *time.timerec* data type. However, the standard library routines allow any 16-bit signed value for the hours *fields*. Note that the *mins* and *secs* fields are still limited to the range 0..59 (and are considered invalid if they are outside this range).

```
duration:
  record;
```

```

secs:int8;
mins:int8;
hours:int16;

endrecord;

```

The *time.OutputFormat* data type controls how the string conversion functions format time values when converting them to strings. This is an enumerated data type with the following values:

```

OutputFormat:  enum
{
    hhmmssAMPM,
    hhmmssAP,
    hhmmss12,
    hhmmss24,

    hhmmAMPM,
    hhmmAP,
    hhmm12,
    hhmm24,

    badTimeFormat
};

```

The Standard Library maintains an internal static variable that keeps track of the current output format (which you can change via the `time.setFormat` function). The various settings affect the output format as follows:

**hhmmssAMPM:** Date is output using a 12-hour clock in the range 01:00:00 to 12:59:59 with an "AM" or "PM" suffix on the time.

**hhmmssAP:** Date conversion uses a 12-hour clock in the range 01:00:00 to 12:59:59 with an "A" or "P" suffix on the time.

**hhmmss12:** Date conversion uses a 12-hour clock in the range 01:00:00 to 12:59:59 with no suffix to denote morning or evening times.

**hhmmss24:** Date conversion uses a 24-hour clock in the range 00:00:00 to 23:59:59.

**hhmmAMPM:** Date is output using a 12-hour clock in the range 01:00 to 12:59 with an "AM" or "PM" suffix on the time.

**hhmmAP:** Date conversion uses a 12-hour clock in the range 01:00 to 12:59 with an "A" or "P" suffix on the time.

**hhmm12:** Date conversion uses a 12-hour clock in the range 01:00 to 12:59 with no suffix to denote morning or evening times.

**hhmm24:** Date conversion uses a 24-hour clock in the range 00:00 to 23:59.

## 37.3 Time Predicates

The functions in this category test times for validity and do other checks on times.

```

time.validate( h:word; m:byte; s:byte );
time.validate( hms:time.timerec );
time._validate( tm:timerec );

```

HLA high-level calling sequence examples:

```

try

    time.validate( someTimeVar );

    anyexception

    // Do something if the time is invalid

endtry;

try

    time.validate( someHour, someMinute, someSecond );

    anyexception

    // Do something if the time is invalid

endtry;

try

    time._validate( someTimeVar );

    anyexception

    // Do something if the time is invalid

endtry;

```

HLA low-level calling sequence examples:

```

push( someTimeVar.time );
call time._validate;

```

The functions in this category test times for validity and do other checks on times.

```

time.isValid( h:word; m:byte; s:byte );
time.isValid( hms:time.timerec );
time._isValid( tm:timerec );

```

HLA high-level calling sequence examples:

```

time.isValid( someTimeVar );
mov( al, timeIsValidVar1 );
time.isValid( someHour, someMinute, someSecond );
mov( al, timeIsValid2 );
time._validate( someTimeVar );
mov( al, timeIsValid3 );

```

HLA low-level calling sequence examples:

```

push( someTimeVar.time );
call time._isValid;
mov( al, timeIsValid3 );

```

## 37.4 Time Conversions

The functions in this category convert time between hours/minutes/second format and an integer specifying some number of seconds, and the functions in this category also perform basic time arithmetic functions such as the addition and subtraction of time.

**#macro unpack( tm, h, m, s )**

This macro takes a *time.timerec* object as its first argument and extracts the *hours*, *mins*, and *secs* fields (zero-extending them to 32 bits) and stores the extract values in the dword *h*, *m*, and *s* arguments (respectively).

HLA macro invocation examples:

```
time.unpack( sometimeVar, hoursVar32, MinutesVar32, SecondsVar32 );
```

**#macro pack( h, m, s, \_tm\_ )**

This macro takes the hours (h), minutes (m), and seconds (s) arguments and packs them into a *time.timerec* object. This macro is very similar to the *date.pack* macro, see the description of that macro for more details about the operation of this macro. Note that if h, m, or s are constant values, this macro will check them to see if they are valid (that is, values in the range 00:00:00 to 23:59:59).

HLA macro invocation examples:

```
time.unpack( hoursVar32, MinutesVar32, SecondsVar32, someTimeVar );
```

**time.durationToSecs( hours:word; mins:byte; secs:byte ); @returns( "eax" );**

This function converts a time span in HHMMSS format to some number of seconds (if HHMMSS is the time of day, then these functions return the time in seconds since midnight). Note that HHMMSS does not have to be a 12-hour or 24-hour clock value. You may specify any number of hours between 0 and 65535, and any number of seconds or minutes between 0 and 255 for this function.

**time.secsToDuration**

```
(
    seconds :uns32;
    var hours :word;
    var mins :byte;
    var secs :byte
);
```

This function converts some number of seconds to a duration (the number of hours, minutes, and seconds) and stores that duration in the *hours*, *mins*, and *seconds* parameters passed by reference. If the number of seconds exceeds 65535 hours, 59 minutes, and 59 seconds, then this function raises an *ex.TimeOverflow* exception.

HLA high-level calling sequence examples:

```
time.secsToDuration( seconds, hoursVar32, MinsVar32, SecsVar32 );
```

HLA low-level calling sequence examples:

```
push( seconds );
pushd( &hoursVar32 );// Assumes hoursVar32 is STATIC
lea( eax, MinsVar32 );// MinsVar32 need not be static
push( eax );
lea( eax, SecsVar32 );// SecsVar32 need not be static
push( eax );
call time.secsToDuration;
```



```
#macro time.toSecs( theTime: time.timerec); @returns( "eax");
#macro time.toSecs( h:uns16; m:byte; s:byte ); @returns( "eax");
time._toSecs( HMS:timerec ); @returns( "eax");
```

These functions convert a time span in HHMMSS format to some number of seconds (if HHMMSS is the time of day, then these functions return the time in seconds since midnight). Technically, these functions and macros don't care if their parameters are valid times (that is, within the range 00:00:00 to 23:59:59), however you should use *time.durationToSecs* when conversion durations (versus valid time of day values) to seconds.

HLA high-level calling sequence examples:

```
time.toSecs( someTimeVar );
mov( eax, numSeconds1 );
time.toSecs( hours, minutes, seconds );
mov( eax, numSeconds2 );
time._toSecs( someTimeVar );
mov( eax, numSeconds3 );
```

HLA low-level calling sequence examples:

```
push( someTimeVar.time );
call time.toSecs;
mov( eax, numSeconds3 );
```

```
time.fromSecs( seconds:uns32; var HMS:time.timerec );
```

This function converts the seconds parameter to an HMS time value. The first parameter must be less than 235,929,600 since this is the maximum time representable by 65535 hours. If the seconds parameter exceeds this value, then *time.secsToHMS* will raise an *ex.TimeOverflow* exception.

HLA high-level calling sequence example:

```
time.fromSecs( seconds, someTimeVar );
```

HLA low-level calling sequence examples:

```
push( seconds );
lea( eax, someTimeVar );// If someTimeVar is non-static
call time.fromSecs;

push( seconds );
pushd( &someTimeVar );// If someTimeVar is static
```

```
time.toUnixTime( DMY:date.daterec; HMS:timerec );
@returns( "edx:eax" );
```

This function converts a Standard Library date and time value to a UNIX/C stdlib date/time value. UNIX/C stdlib time values are specified as the number of seconds since midnight, Jan 1, 1970. This function raises an *ex.InvalidDate* exception if the DMY parameter specifies a date prior to Jan 1, 1970.

This function returns a 64-bit value. Most UNIX systems and C standard library packages currently specify a "time\_t" object as a 32-bit signed integer. This data type will fail to properly maintain dates sometime during the year 2038. Newer systems define *time\_t* as an unsigned 32-bit integer, thereby doubling the effective range of the date. Nevertheless, the *date.daterec* data type can represent dates outside the range of even a 32-bit unsigned integer, so this function returns a 64-bit value in EDX:EAX. If you need to work with a 32-bit value, simply ignore the value returned in EDX.

HLA high-level calling sequence example:

```
time.toUnixTime( someDate, someTime );
mov( eax, (type dword unixTimeVar));
mov( edx, (type dword unixTimeVar[4]));// Assuming it's 64 bits.
```

HLA low-level calling sequence examples:

```
push( someDate.date );
push( someTime.time );
call time.toUnixTime;
mov( eax, (type dword unixTimeVar));
mov( edx, (type dword unixTimeVar[4]));// Assuming it's 64 bits.
```

#### **time.fromUnixTime**

```
(
    unixTime      :qword;
    var HMS       :timerec;
    var DMY       :date.daterec
);
```

This function converts the UNIX/C standard library *time\_t* object passed in *unixTime* to HLA Standard Library *date.daterec* (*DMY*) and *time.timerec* (*HMS*) objects. Note that the *time\_t* type on most Unix systems (and in the C standard library) is a 32-bit value whereas this function expects a 64-bit value. If working with actual 32-bit *time\_t* values, simply zero extend them to 64 bits before calling this function.

HLA high-level calling sequence example:

```
time.fromUnixTime( unixDateTime, someDate, someTime );
```

HLA low-level calling sequence examples:

```
// If the unix date/time on your system is 32 bits:

pushd( 0 );
push( (type dword unixTimeVar));
pushd( &someDate.date );// Assumes someDate.date and
pushd( &someTime.time );// someTime.time are STATIC
call time.fromUnixTime;

// If the unix date/time on your system is 64 bits:

push( (type dword unixTimeVar[4]));
push( (type dword unixTimeVar));
lea( eax, someDate.date );// Assumes someDate.date and
push( eax );               // someTime.time are not STATIC
lea( eax, someTime.time );
push( eax );
call time.fromUnixTime;
```

```
time.toWinFileTime( DMY:date.daterec; HMS:timerec );
@returns( "edx:eax" );
```

Windows file times are 64-bit values that represent the number of 100 nanosecond periods since midnight, Jan 1, 1601. This function converts a Standard Library date and time value (passed in the *DMY* and *HMS* parameters) to a Windows file time and returns that value in the EDX:EAX register pair. Because HLA time values only maintain seconds precision, the resulting value will have a granularity of one second. If you actually need to create a value with finer granularity, add the number of 0.1 microseconds to the result that *time.toWinFileTime* returns. This function raises an *ex.InvalidDate* exception if the Standard library date is less than Jan 1, 1601.

HLA high-level calling sequence example:

```
time.toWinFileTime( someDate, someTime );
mov( eax, (type dword win32TimeVar));
mov( edx, (type dword win32TimeVar[4]));
```

HLA low-level calling sequence examples:

```
push( someDate.date );
push( someTime.time );
call time.toWinFileTime;
mov( eax, (type dword win32TimeVar));
mov( edx, (type dword win32TimeVar[4]));
```

```
time.fromWinFileTime
(
    winTime :qword;
    var HMS :timerec;
    var DMY :date.daterec
);
```

This function converts a Windows file time to a Standard Library date and time. This function stores the resulting date in the *DMY* parameter and the time to the *HMS* parameter (both passed by reference). Because Windows file times provide 100ns precision whereas the Standard Library functions only work with 1 sec precision, this function rounds the Windows time to the nearest second during the conversion (specifically, if there are 0.5 or more fractional seconds, this function bumps up the seconds value by one).

HLA high-level calling sequence example:

```
time.fromWinFileTime( win32DateTime, someDate, someTime );
```

HLA low-level calling sequence example:

```
push( (type dword win32DateTime[4]));
push( (type dword win32DateTime));
lea( eax, someDate.date );// Assumes someDate.date and
push( eax );              // someTime.time are not STATIC
lea( eax, someTime.time );
push( eax );
call time.fromWinFileTime;
```

## 37.5 Time Arithmetic

```
time.secsBetweenTimes( time1:timerec; time2:timerec ); @returns( "eax" );
```

This function computes the number of seconds between the two times passed as parameters. It returns the value in the EAX register. Note that this is the absolute value of their difference, so the relative sizes of the two operands is immaterial. Both times must be valid Standard Library *timerec* values in the range 00:00:00..23:59:59 or this function will raise an *ex.InvalidTime* exception.

HLA high-level calling sequence example:

```
time.secsBetweenTimes( someTime1, someTime2 );
mov( eax, secondsBetween );
```

HLA low-level calling sequence example:

```

    push( someTime1.time );
    push( someTime2.time );
    call time.secsBetweenTimes;
    mov( secondsBetween );

```

**time.subHours( hours:uns32; var HMS:timerec ); @returns( "eax" );**

This function subtracts the number of *hours* from the *timerec* object passed by reference as the second parameter (*HMS*). This function returns in EAX the number of days "borrowed" during the calculation (that is, for each transition past midnight during this calculation, the calculation "borrows" one day).

HLA high-level calling sequence example:

```
time.subHours( hours, someTime );
```

HLA low-level calling sequence examples:

```

    push( hours );
    pushd( &someTime );// Assuming someTime is STATIC
    call time.subHours;

    push( hours );
    lea( eax, someTime );// Assuming someTime is not STATIC
    push( eax );
    call time.subHours;

```

**time.subMins( minutes:uns32; var HMS:timerec ); @returns( "eax" );**

This function subtracts the number of *minutes* from the *timerec* object passed by reference as the second parameter (*HMS*). This function returns in EAX the number of days "borrowed" during the calculation (that is, for each transition past midnight during this calculation, the calculation "borrows" one day).

HLA high-level calling sequence example:

```
time.subMins( minutes, someTime );
```

HLA low-level calling sequence examples:

```

    push( minutes );
    pushd( &someTime );// Assuming someTime is STATIC
    call time.subMins;

    push( minutes );
    lea( eax, someTime );// Assuming someTime is not STATIC
    push( eax );
    call time.subMins;

```

**time.subSecs( seconds:uns32; var HMS:timerec ); @returns( "eax" );**

This function subtracts the number of *seconds* from the *timerec* object passed by reference as the second parameter (*HMS*). This function returns in EAX the number of days "borrowed" during the calculation (that is, for each transition past midnight during this calculation, the calculation "borrows" one day).

HLA high-level calling sequence example:

```
time.subSecs( seconds, someTime );
```

HLA low-level calling sequence examples:

```

push( seconds);
pushd( &someTime );// Assuming someTime is STATIC
call time.subSecs;

push( seconds);
lea( eax, someTime );// Assuming someTime is not STATIC
push( eax );
call time.subSecs;

```

**time.addHours( hours:uns32; var HMS:timerec ); @returns( "eax" );**

This function adds the number of *hours* to the *timerec* object passed by reference as the second parameter (*HMS*). This function returns in EAX the number of days skipped during the calculation (that is, for each transition past midnight during this calculation, the calculation "skips" one day).

HLA high-level calling sequence example:

```
time.addHours( hours, someTime );
```

HLA low-level calling sequence examples:

```

push( hours );
pushd( &someTime);// Assuming someTime is STATIC
call time.addHours;

push( hours );
lea( eax, someTime );// Assuming someTime is not STATIC
push( eax );
call time.addHours;

```

**time.addMins( minutes:uns32; var HMS:timerec ); @returns( "eax" );**

This function adds the number of *minutes* to the *timerec* object passed by reference as the second parameter (*HMS*). This function returns in EAX the number of days skipped during the calculation (that is, for each transition past midnight during this calculation, the calculation "skips" one day).

HLA high-level calling sequence example:

```
time.addMins( minutes, someTime );
```

HLA low-level calling sequence examples:

```

push( minutes );
pushd( &someTime );// Assuming someTime is STATIC
call time.addMins;

push( minutes );
lea( eax, someTime );// Assuming someTime is not STATIC
push( eax );
call time.addMins;

```

**time.addSecs( seconds:uns32; var HMS:timerec ); @returns( "eax" );**

This function adds the number of *seconds* to the *timerec* object passed by reference as the second parameter (*HMS*). This function returns in EAX the number of days skipped during the calculation (that is, for each transition past midnight during this calculation, the calculation "skips" one day).

HLA high-level calling sequence example:

```
time.addSecs( seconds, someTime );
```

HLA low-level calling sequence examples:

```
push( seconds);
pushd( &someTime );// Assuming someTime is STATIC
call time.addSecs;

push( seconds);
lea( eax, someTime );// Assuming someTime is not STATIC
push( eax );
call time.addSecs;
```

## 37.6 Reading the Current System Time

```
time.curTime( var theTime: time.timerec );
```

This returns the local time (provided by the system clock) in the specified time variable.

HLA high-level calling sequence example:

```
time.curTime( someTime );
```

HLA low-level calling sequence examples:

```
pushd( &someTime );// Assuming someTime is STATIC
call time.curTime;

lea( eax, someTime );// Assuming someTime is not STATIC
push( eax );
call time.curTime;
```

```
time.utcTime( var theTime: time.timerec );
```

This returns the UTC time (the current GMT time provided by the system clock) in the specified time variable.

HLA high-level calling sequence example:

```
time.utcTime( someTime );
```

HLA low-level calling sequence examples:

```
pushd( &someTime );// Assuming someTime is STATIC
call time.utcTime;

lea( eax, someTime );// Assuming someTime is not STATIC
push( eax );
call time.utcTime;
```

## 37.7 Time String Conversions and Output

```
time.setFormat( f:OutputFormat );
```

This function sets the global system time conversion value. The parameter must be one of the following `time.OutputFormat` enumerated values:

```
hhmmssAMPM
hhmmssAP
hhmmss12
hhmmss24
hhmmAMPM
hhmmAP
hhmm12
hhmm24
```

The first four constants tell the time/string conversion routines to emit hours, minutes, and seconds in a "00:00:00" format, the last four output only the hours and minutes in a "00:00" format. The *hhmmssAMPM* and *hhmmAMPM* constants emit a 12-hour time format with either "am" or "pm" appended to the string to denote midnight to noon or noon to midnight. The *hhmmssAP* and *hhmmAP* formats do the same, except that they only display an "a" or a "p" after the time. The *hhmmss12* and *hhmm12* formats display a 12-hour time with no indication of which half of the day the time represents. The *hhmmss24* and *hhmm24* formats specify a 24-hour time.

The *time.toString* and *time.a\_toString* functions use the value of the global time format to determine how they convert a Standard Library `timerec` value to a string.

```
time.toString( HMS:timerec; dest:string );
```

This function converts the *HMS* parameter to a string using the format specified by the global *OutputFormat* variable (set by the *time.setFormat* function). The destination string must have sufficient storage associated with it or this function will raise an exception. This function will also raise an *ex.InvalidTime* exception if *HMS* contains an invalid time.

HLA high-level calling sequence example:

```
time.toString( someTime, destStr );
```

HLA low-level calling sequence example:

```
push( someTime.time );
push( destStr );
call time.toString;
```

```
time.a_toString( HMS:timerec ); @returns( "eax" );
```

This function is similar to *time.toString* except you don't supply a destination string. Instead, this function allocates storage for the string on the heap. Note that it is the caller's responsibility to free this storage when the caller is done with the string (i.e., by calling *str.free*).

HLA high-level calling sequence example:

```
time.a_toString( someTime );
mov( eax, destStr );
```

HLA low-level calling sequence example:

```
push( someTime.time );
call time.a_toString;
mov( eax, destStr );
```

## 37.8 Time Class Types

```
#include( "dtClass.hhf" )
```

Note: the `stdlib.hhf` header file does not include `dtClass.hhf`. If you want to use the time class data types you will need to explicitly include the `dtClass.hhf` header file.

For those who prefer an object-oriented programming approach, the Standard Library provides the ability to create time class data types. The Standard Library provides two predefined time class types: *timeClass\_t* and *virtualTimeClass\_t*. The difference between these two types is that the *timeClass\_t* type uses static procedures for all the time functions whereas *virtualTimeClass\_t* uses virtual methods. In certain cases, using the *timeClass\_t* data type is more efficient than using *virtualTimeClass\_t* because you only link in the class functions you actually call. However, you lose the ability to make polymorphic method calls when using the *timeClass\_t*. For more details on the differences between these two class types, please see the discussion of the `dtClass.make_timeClass` macro appearing later in this section. This section will use the phrase "time class" to mean any class created by the `make_timeClass` macro, including the *timeClass\_t* and *virtualTimeClass\_t* data types.

The time class types provide two data fields:

```
var
    theTime :time.timerec;
    timeFmt :time.OutputFormat;
```

The first field, *theTime*, holds the time value associated with the time object. This is the standard *time.timerec* date type described earlier in this document. Note that you can pass this field to any of the standard date and time functions that expect a *time.timerec* value.

The second field, *timeFmt*, specifies the output format when using the time class string conversion routines. Note that only the time class string conversion routines respect the value of this field; if you pass *theTime* directly to a time function that takes a *time.timerec* argument, that function will use the system-wide global time format rather than the object's *timeFmt* value.

**Thread Safety Issue:** Although each time object has its own *timeFmt* field, this does not make the use of time class objects thread safe. When converting *theTime* to a string, the time class functions save the global format value, copy *timeFmt* to the global variable, call the time functions to do the string conversion, and then restore the original global value. If a thread is suspended during this activity then any time/string conversions during this suspension may use an incorrect format value. This issue will be corrected in a later version of the Standard Library. For now, you must manually protect all time/string conversions if you perform such conversions in multiple threads in your application.

Of course, you may create a derived class from either *timeClass\_t* or *virtualTimeClass\_t* (or create a brand new time class using the `dtClass.make_timeClass` macro) and add any other fields you like to that new time class. One suggestion for such a class is to pad the data fields to a multiple of four bytes. Currently, the *timeClass\_t* and *virtualTimeClass\_t* objects consumes nine bytes of storage (five bytes for the three fields above plus four bytes for the VMT pointer). For performance reasons, you might want to extend the size of the data storage to 12 or even 16 bytes. Another suggestion might be to add a *Separator* field that specifies the hours/minutes/seconds separator character when converting a time to a string; of course, you'll need to override the *toString* and *a\_toString* methods to achieve this.

## 37.9 Time Class Methods/Procedures

In most HLA classes, there are two types of functions: (static) procedures and (dynamic) methods (there are also iterators, but the time classes do not use iterators so we will ignore that here). The only difference between a method and a procedure is how the program actually calls the function: the program calls procedures directly, it calls methods indirectly through an entry in the virtual method table (VMT). Static procedure calls are very efficient, but you lose the object-oriented benefits of polymorphism when you define a function as a static procedure in a class. Methods, on the other hand, fully support polymorphic calls, but introduce some efficiency issues. Let's consider those issues here.

First of all, unlike static procedures, your program will link in all methods defined in your program *even if you don't explicitly call those methods in your program*. Because the call is indirect, there really is no way for the assembler and linker to determine whether you've actually called the function, so it must assume that you do call it and the linker links in the code for each method in the class. This can make your program a little larger because it may be including several time class functions that you don't actually call. For large applications, the amount of extra storage required by linking in all the time functions is inconsequential, but if you don't like linking in code that the program will never call, specifying virtual methods for all the time functions may annoy you.



The second efficiency issue concerning method calls is that they use the EDI register to make the indirect call (static procedure calls do not disturb the value in EDI). Therefore, you must ensure that EDI is free and available before making a virtual method call, or take the effort to preserve EDI's value across such a call.

A third, though extremely minor, efficiency issue concerning methods is that the class' VMT will need an extra entry in the virtual method table. As this is only four bytes per class (not per object), this probably isn't much of a concern.

The predefined *timeClass\_t* and *virtualTimeClass\_t* functions differ in how they define the functions appearing in the class types. The *timeClass\_t* type uses static procedures for all functions, the *virtualTimeClass\_t* type uses methods for all class functions (except the constructor, *create*, because constructors are always static procedures). Therefore, *timeClass\_t* data types will make direct calls to all the functions (and only link in the procedures you actually call); however, *timeClass\_t* objects do not support function polymorphism in derived classes. The *virtualTimeClass\_t* type does support polymorphism for all the class methods, but whenever you use this data type you will link in all the methods (even if you don't call them all) and calls to these methods will require the use of the EDI register.

It is important to understand that *timeClass\_t* and *virtualTimeClass\_t* are two separate types. Neither is derived from the other. Nor are the two types compatible with one another. You should take care not to confuse objects of these two types if you're using both types in the same program.

## 37.10 Creating New Time Class Types

As it turns out, the only difference between a method and a procedure (in HLA) is how that method/procedure is called. The actual function code is identical regardless of the declaration (the reason HLA supports method and procedure declarations is so that it can determine how to populate the VMT and to determine how to call the function). By pulling some tricks, it's quite possible to call a procedure using the method invocation scheme or call a method using a direct call (like a static procedure). The Standard Library time class module takes advantage of this trick to make it possible to create new time classes with a user-selectable set of procedures and methods. This allows you to create a custom time type that uses methods for those functions you want to override (as methods) and use procedures for those functions you don't call or will never override (as virtual methods). Indeed, the *timeClass\_t* and *virtualTimeClass\_t* time types were created using this technique. The *timeClass\_t* data type was created specifying all functions as procedures, the *virtualTimeClass\_t* data type was created specifying all functions, except *create*, as methods. By using the *dtClass.make\_timeClass* macro, you can create new time data types that have any combination of procedures and methods.

```
dtClass.make_timeClass( className, "<list of methods>" )
```

*dtClass.make\_timeClass* is a macro that generates a new data type. As such, you should only invoke this macro in an HLA type declaration section. This macro requires two arguments: a class name and a string containing the list of methods to use in the new data type. The method list string must contain a sequence of method names from the following list:

```
create
curTime
utcTime
addSecs
addMins
addHours
subSecs
subMins
subHours
fromSecs
toSecs
isValid
validate
difference
secsBetweenTimes
toString
a_toString
```

Here is *dtClass.make\_timeClass* macro invocation that creates the *virtualTimeClass\_t* type; note that the *create* function is always a static procedure and its name must not appear in the list of method names:

```

type
    dtClass.make_timeClass
    (
        virtualTimeClass_t,
        "curTime "
        "utcTime "
        "addSecs "
        "addMins "
        "addHours "
        "subSecs "
        "subMins "
        "subHours "
        "fromSecs "
        "toSecs "
        "isValid "
        "validate "
        "difference "
        "secsBetweenTimes "
        "toString "
        "a_toString "
    );

```

(For those unfamiliar with the syntax, HLA automatically concatenates string literals that are separated by nothing but whitespace; therefore, this macro contains exactly two arguments, the *virtualTimeClass\_t* name and a single string containing the concatenation of all the strings above.)

From this macro invocation, HLA creates a new data type using methods for each of the names appearing in the string argument. If a particular time function's name is not present in the *dtClass.make\_timeClass* macro invocation, then HLA creates a static procedure for that function. As a second example, consider the declaration of the *timeClass\_t* data type (which uses static procedures for all the time functions):

```

type
    dtClass.make_timeClass( timeClass_t, " " );

```

Because the function string does not contain any of the time function names, the *dtClass.make\_timeClass* macro generates static procedures for all the time functions.

The *timeClass\_t* type is great if you don't need to create a derived time class that allows you to polymorphically override any of the time functions. If you do need to create methods for certain functions and you don't mind the overhead of a virtual method call, the *virtualTimeClass\_t* makes all the functions. Probably 99% of the time you won't be calling the time functions very often, so the overhead of using method invocations for all time functions is irrelevant. In those rare cases where you do need to support polymorphism for a few time functions but don't want to link in the entire set of time functions, or you don't want to pay the overhead for indirect calls to functions that are never polymorphic, you can create a new time class type that specifies exactly which functions require polymorphism.

For example, if you want to create a time class that overrides the definition of the *fromSecs* and *toSecs* functions, you could declare that new type thusly:

```

type
    dtClass.make_timeClass
    (
        myTimeClass,
        "fromSecs"
        "toSecs"
    );

```

This new class type (*myTimeClass*) has two methods, *fromSecs* and *toSecs*, and all the other time functions are static procedures. This allows you to create a derived class that overloads the *fromSecs* and *toSecs* methods and access those methods when using a generic *myTimeClass* pointer, e.g.,

```

type
  derivedMyTimeClass :
    class inherits( myTimeClass );

        override method fromSecs;
        override method toSecs;

    endclass;

```

It is important for you to understand that types created by *dtClass.make\_timeClass* are base types. They are not derived from any other class (e.g., *virtualTimeClass\_t* is not derived from *timeClass\_t* or vice-versa). The types created by the *dtClass.make\_timeClass* macro are independent and incompatible types. For this reason, you should avoid using different base time class types in your program. Pick (or create) a base time class and use that one exclusively in an application. You'll avoid confusion by following this rule.

For the sake of completeness, here are the macros that the Standard Library uses to create time data types:

```

namespace dtClass;

// The following macro allows us to turn a class function
// into either a method or a procedure based on the
// presence of "funcName" within a list of method names
// passed to the class generating macro.

#macro function( funcName );

    #if( @index( methods, 0, @string:funcName) = -1 )

        procedure funcName

    #else

        method funcName

    #endif

#endmacro

// make_timeClass -
//
// This macro is used to create a base time class.
// The first parameter is the name of the class to create.
// The second parameter is a string listing the 'function'
// names that you want converted to a class method (if not
// present, it will be a class procedure).

#macro make_timeClass( className, methods );

    className:
    class

        var

            theTime :time.timerec;
            timeFmt :time.OutputFormat;

        procedure create;
            @external( "TIMECLASS_CREATE" );

        dtClass.function( curTime );
            @external( "TIMECLASS_CURTIME" );

```

```

dtClass.function( utcTime );
    @external( "TIMECLASS_UTCTIME" );

dtClass.function( addSecs )( seconds:uns32 );
    @external( "TIMECLASS_ADDSECS" );

dtClass.function( addMins )( minutes:uns32 );
    @external( "TIMECLASS_ADDMINS" );

dtClass.function( addHours )( hours:uns32 );
    @external( "TIMECLASS_ADDHOURS" );

dtClass.function( subSecs )( seconds:uns32 );
    @external( "TIMECLASS_SUBSECS" );

dtClass.function( subMins )( minutes:uns32 );
    @external( "TIMECLASS_SUBMINS" );

dtClass.function( subHours )( hours:uns32 );
    @external( "TIMECLASS_SUBHOURS" );

dtClass.function( fromSecs )( seconds:uns32 );
    @external( "TIMECLASS_FROMSECS" );

dtClass.function( toSecs );
    @returns( "eax" );
    @external( "TIMECLASS_TOSECS" );

dtClass.function( isValid );
    @returns( "al" );
    @external( "TIMECLASS_ISVALID" );

dtClass.function( validate );
    @external( "TIMECLASS_VALIDATE" );

dtClass.function( difference )( var time2:className );
    @returns( "eax" );
    @external( "TIMECLASS_DIFFERENCE" );

dtClass.function( secsBetweenTimes )( time2:time.timerec );
    @returns( "eax" );
    @external( "TIMECLASS_SECSBETWEENTIMES" );

dtClass.function( toString )( dest:string );
    @external( "TIMECLASS_TOSTRING" );

dtClass.function( a_toString );
    @external( "TIMECLASS_A_TOSTRING" );

endclass;

#endmacro

end dtClass;

```

If you look closely at the *make\_timeClass* macro, you'll notice that it maps all the functions, be they methods or procedures, to the *timeClass\_t* names (which are all procedures, if you look at the source code for these functions). As noted earlier, the function code for methods and procedures is exactly the same, only the call to a given function is different based on whether it is a method or a procedure. Therefore, the *dtClass.make\_timeClass* macro maps all functions to the same set of procedures. Therefore, if you do create and use multiple date classes in the same application, the linker will only link in one set of routines (unless, of course, you overload some methods, in which case the linker will link in your new functions as well as the original *timeClass\_t* set).

## 37.11 Time Class Functions

The time class type supports most of the functions associated with the time type. The main difference is that the time class functions operate directly on the time object rather than on a time value you pass as a parameter. For this reason, there aren't any macros that overload the time function parameter lists.

In the following function descriptions, the symbol *<object>* is used to specify a time class object or a pointer to a time class object. Note that class invocations of static procedures (e.g., *timeClass\_t.isValid*) are illegal with the single exception of the constructor (the *create* procedure). If you call a time class procedure directly, the system will raise an exception (as ESI, which should be pointing at the object's data, will contain NULL).

Note: because the syntax varies from declaration to declaration, the following sections do not provide examples of calling these functions, please see the HLA documentation under object-oriented programming or *The Art of Assembly Language Programming* for more details.

***<object>.create();***

The *<name>.create* procedure is the object constructor. This is the only function that you may call using a class name rather than an object name. For example, *timeClass\_t.create();* is a perfectly legitimate constructor call. As is the convention for HLA class constructors, if you call a class constructor directly (using the class name rather than an object name), the time class constructor will allocate storage for a new time class object on the heap and return a pointer to the new object in ESI. Once the storage is allocated (or if you specify the name of a previously-allocated object rather than the class name), the time class constructor will initialize all the fields of the object to reasonable values (in particular, the constructor initializes the VMT pointer, initializes *theTime* to a valid time (00:00:00), and sets up the *theFmt* field with a default value).

If you create a derived time class and add new data fields to the data type, you should override the *create* procedure and initialize those new fields in the overridden procedure. See the HLA documentation or *The Art of Assembly Language* for more details on derived classes and overriding constructors.

***<object>.validate();***

The *<object>.validate* function checks the validity of an object's *theTime* field. It raises an *ex.InvalidTime* exception if the object's *theTime* field contains an invalid value (hours outside the range 0..23 or minutes/seconds outside the range 0..59). See *time.validate* for more details.

***<object>.isValid(); @returns( "al" );***

The *<object>.isValid* function checks the validity of an object's *theTime* field. It returns true (in AL, zero-extended into EAX) if *theTime* field contains a valid time value, it returns false otherwise. See *time.isValid* for more details.

***<object>.toSecs(); @returns( "eax" );***

This function converts the object's *theTime* field (in HH:MM:SS format) to the number of seconds since midnight. This function returns the result in the EAX register. See the *time.toSecs* function description for more details.

***<object>.fromSecs( seconds:uns32 );***

This function converts the parameter value (*seconds*, the number of seconds since midnight) into a standard library compatible HH:MM:SS time format and stores the result in the object's *theTime* field. This function returns the number of overflow days (that is, the number of 24-hour periods) in the EAX register, the value this

function stores into *theTime* is always a valid time between 00:00:00 and 23:59:59. See the *time.toSecs* function description for more details.

```
<object>.secsBetweenTimes( otherTime:timerec ); @returns( "eax" );
```

This function computes the number of seconds between the object's *theTime* value and a *time.timerec* value you pass as a parameter. It returns the difference, in seconds, in the EAX register. See the *time.secsBetweenTimes* function for more information.

```
<object>.difference( var otherTime:<object's class> ); @returns( "eax" );
```

This function computes the number of seconds between the object's *theTime* value and same field in the object you pass as a parameter. It returns the difference, in seconds, in the EAX register. The type of the parameter object must be the same type as *<object>* (i.e., *timeClass t*, *virtualTimeClass t*, or whatever other time class you've created and defined *<object>* to be). See the *time.secsBetweenTimes* function for more information.

```
<object>.subHours( hours:uns32 );
```

This function subtracts the number of hours specified by the parameter from the object's *theTime* field. See the *time.subHours* function for more information.

```
<object>.subMins( hours:uns32 );
```

This function subtracts the number of minutes specified by the parameter from the object's *theTime* field. See the *time.subMins* function for more information.

```
<object>.subSecs( hours:uns32 );
```

This function subtracts the number of seconds specified by the parameter from the object's *theTime* field. See the *time.subSecs* function for more information.

```
<object>.addHours( hours:uns32 );
```

This function adds the number of hours specified by the parameter to the object's *theTime* field. See the *time.addHours* function for more information.

```
<object>.addMins( minutes:uns32 );
```

This function adds the number of minutes specified by the parameter to the object's *theTime* field. See the *time.addMins* function for more information.

```
<object>.addSecs( seconds:uns32 );
```

This function adds the number of seconds specified by the parameter to the object's *theTime* field. See the *time.addSecs* function for more information.

```
<object>.curTime();
```

This stores the local time (provided by the system clock) in the object's *theTime* field. See the *time.curTime* function for additional details.

```
<object>.utcTime();
```

This stores the UTC time (the current GMT time provided by the system clock) in the objects *theTime* field. See the *time.utcTime* function for additional details.

```
<object>.toString( dest:string );
```

This function converts the object's *theTime* field to a string using the object's *OutFmt* field to guide the conversion. This function stores the character data in the string object pointed at by the *dest* parameter (there must be sufficient space allocated for the string or the function will raise an exception). See the *time.toString* function for more information.

```
<object>._a_toString( HMS:timerec ); @returns( "eax" );
```

This function converts the object's *theTime* field to a string using the object's *OutFmt* field to guide the conversion. This function allocates storage for the resultant string and returns a pointer to this new string in EAX. It is the caller's responsibility to deallocate the storage associated with this string when the caller is done with it. See the *time.\_a\_toString* function for more information.

Because the time class includes an *"\_a\_toString"* function, you may print time object values using *stdout.put* and similar *\*.put* Standard Library functions. Note that those functions automatically deallocate the storage associated with the string created by *<object>.\_a\_toString*.





## 38 Timer Class and Module (timer.hhf)

The HLA Timer module provides a set of routines that let you time events with millisecond precision.

**Note:** Like documentation for most standard library modules that are based on an HLA class, this document does not provide examples of low-level calls to the timer functions. If you're interested in making low-level machine instruction calls to the methods in the timer class, please consult the HLA documentation concerning classes and objects.

**A Note About Thread Safety:** The timer module maintains various values within each object. If you attempt to manipulate the same object from different threads in a multi-threaded application, you may get inconsistent results. Therefore, you should only call the procedures and methods for a particular timer object from one thread or you must explicitly control access to those methods to prevent concurrent execution of the same object's methods from different threads. Note that you may call the methods for *different* timer objects from different threads.

### 38.1 Timer Module

To use the timer functions in your application, you will need to include one of the following statements at the beginning of your HLA application:

```
#include( "timer.hhf" )
or
#include( "stdlib.hhf" )
```

### 38.2 Timer Class/Data Structure

The Timer module is actually a class with the following definition:

```
timer_t: class
    var
        Accumulated:    qword;

        DateStarted:    date.daterec;
        TimeStarted:    time.timerec;
        msStarted:      uns32;

        DateStopped:    date.daterec;
        TimeStopped:    time.timerec;
        msStopped:      uns32;

        Running:        boolean;
        Valid:           boolean;

    procedure create;    external;

    method start;        external;
    method restart;      external;
    method stop; @returns( "edx:eax" ); external;
    method checkPoint; @returns( "edx:eax" ); external;

endclass;
```

Don't forget that the *timer\_t* class, like all class objects, will modify the values of the ESI and EDI registers whenever you call a class procedure or method. So don't expect values in ESI or EDI to be preserved across the calls in this module.

## 38.3 Timer Operation

The `timer_t` class maintains an accumulation of time. When you create a class object, or when you call the `timer_t.start` method, the system initializes this 64-bit unsigned integer value to zero. When you call the `timer_t.start` method, the system notes the point at which you called the method so it can compute the amount of accumulated time when you call `timer_t.stop` or `timer_t.checkPoint` at some point in the future. It is important that you realize that the class' `timer_t.Accumulated` field does *not* contain a real-time representation of the elapsed time. When you call `timer_t.stop`, the object will compute the amount of elapsed time since the call to `timer_t.start` and will update `timer_t.Accumulated` with this value. So to time a simple sequence of events, you would first call `timer_t.start`, do whatever it is that you want to time, and then call `timer_t.stop` when you're finished with the events you want to time. On return from `timer_t.stop`, the EDX:EAX register pair will contain the 64-bit elapsed time value, or you can retrieve the value from the object's `timer_t.Accumulated` field.

If you would like to compute the current elapsed time during some timing sequence, but you do not want to stop the timing operation, you can call the `timer_t.checkPoint` method. This method will update the `timer_t.Accumulated` field with the elapsed time up to that point without stopping the timer operation. The `timer_t.checkPoint` function call will also return the total accumulated time in the EDX:EAX register pair. The timer will continue running until you call the `timer_t.stop` method at some point in the future. Note that you may call `timer_t.checkPoint` as many times as you like between the `timer_t.start` and `timer_t.stop` method calls. Note, however, that you may only call `timer_t.checkPoint` while the timer is actually running.

For more complex timing applications, it is possible to start, stop, and restart the timer without resetting the accumulated value to zero. Restarting the timer after calling `timer_t.stop` is possible by calling the `timer_t.restart` method. The `timer_t.restart` method is functionally equivalent to `timer_t.start` except that it doesn't zero out the `timer_t.Accumulated` field. When you call `timer_t.stop` after a `timer_t.restart` method invocation, the `timer_t.accumulated` field is updated with the sum of its previous value plus the measured time between the `timer_t.restart` and `timer_t.stop` calls. Of course, you can make multiple calls to the `timer_t.restart/timer_t.stop` methods to accumulate time over longer periods.

## 38.4 Timer Class Fields

### `timer_t.Accumulated`

This field contains the computed time in milliseconds. This field is only valid if the `timer_t.Valid` field contains true. If `timer_t.Running` contains true, then the timer is still running and the `timer_t.Accumulated` field contains the number of milliseconds at the last `timer_t.checkPoint` or `timer_t.restart` operation.

```
timer_t.DateStarted
timer_t.TimeStarted
timer_t.msStarted
timer_t.DateStopped
timer_t.TimeStopped
timer_t.msStopped
```

These are internal variables to the class. You should not modify their values nor should you read their values and use them for anything.

### `timer_t.Running`

This boolean variable indicates that the timer object is currently timing some event. You may read this variable but you should not modify its value.

## 38.5 Timer Procedures and Methods

```
procedure timer_t.create; @returns( "esi" );
```

This is the constructor for the class. If you call it via "`someObjectName.create()`;" then this static class procedure will initialize the fields of the specified object. If you call it via "`timer_t.create()`;" then `timer_t.create` will dynamically allocate storage for a `timer_t` object and initialize that storage. This call will return a pointer to the new object in the ESI register.

HLA high-level calling sequence examples:

```
timer_t.create();
mov( esi, timerPtrVar );

timerClassVar.create();
```

#### **method timer\_t.start;**

This method will initialize the timer so it can begin timing some sequence. Note that this call will set the *timer\_t.Running* field to true and the *timer\_t.Valid* field to false. Use the *timer\_t.stop* method call to stop the timing operation. This call will also initialize the *timer\_t.Accumulated* field to zero. Calling this method on a timer that is already running will reset the accumulated time to zero. See *timer\_t.restart* if you want to start the timer running without clearing the *timer\_t.Accumulated* field.

HLA high-level calling sequence examples:

```
timer_t.create();
mov( esi, timerPtrVar );
.
.
.
timerPtrVar.start();
.
.
.
timerPtrVar.stop();
mov( edx:eax, qwordTimerValue );// Accumulated time
```

#### **method timer\_t.stop; @returns( "edx:eax" );**

This method will stop the timer accumulation and returns the accumulated time in EDX:EAX. This call sets *timer\_t.Valid* to true and *timer\_t.Running* to false.

#### **method timer\_t.restart;**

This method restarts the timer after you've stopped the timing via *timer\_t.stop*. Note that the result accumulated will be the sum of the previous accumulation plus the new time.

#### **method timer\_t.checkPoint;**

This computes the current time in *timer\_t.Accumulated* without stopping the timer. That is, *timer\_t.Valid* will be set to true and *timer\_t.Running* will remain true.



## 39 Zero-terminated String Functions (zstring.hhf)

Although HLA's string format is more efficient (with respect to speed) than the zero-terminated string format that languages like C, C++, and Java use, HLA programs must often interact with code that expects zero-terminated strings. Examples include HLA (assembly) code you like with C/C++/Java programs and calls you make to operating systems like Windows and Linux (that expect zero terminated strings). Therefore, the HLA Standard Library provides a limited amount of support for zero-terminated strings so it can efficiently interact with external code that requires such strings.

When passing read-only string data to some code that expects a zero-terminated string, HLA's string format is upwards compatible with zero-terminated strings. No conversion is necessary. An HLA string variable holds the address of a sequence of characters that end with a zero byte (the zero-terminated format). So as long as the code you're calling doesn't attempt to write any data to the string object, you can pass HLA string objects to functions and procedures that expect zero-terminated strings.

If the procedure or function you're calling stores data into a destination string variable, then you generally should not pass an HLA string to that function. There are two problems with this: first, the function does not check the HLA string's maximum length field to ensure that string overflow does not occur; second, the external function does not properly set the HLA string's length field before returning. Furthermore, the external code may create it's own string data in some buffer and does not even allocate space for HLA's maximum length and dynamic length fields. To workaround these limitations, HLA provides various procedures in the Standard Library that manipulate zero-terminated strings so your programs can effectively communicate with external code that operates on such strings.

Before describing the support functions that HLA provides for zero-terminated strings, it's probably worthwhile to first discuss how one writes code that comfortably co-exists with such strings. As noted above, there are three major problems one must deal with when external code processes zero-terminated strings. We'll deal with these issues one at a time.

The first problem is that the external code does not check the maximum string length field before writing character data to a string object. Therefore, the external code cannot determine if a buffer overflow will occur when that function extends the string's length. Algorithms that depend upon the string function raising an exception when a buffer overflow occurs will not work properly when calling external code that manipulates zero-terminated strings. The solution to this problem is the same as the solution in C and C++: the programmer must take the responsibility of ensuring that there is sufficient buffer space available to hold the string the external function produces. Exactly how much space you must allocate as a maximum varies on a call by call basis, but usually you can pick a sufficiently large value that is safe and preallocate storage for an HLA string whose maximum length satisfies the program's requirements. Note that most operating system API functions that return variable length strings will let you specify a maximum length parameter so the OS will not overflow your string buffer; well-written library routines and other code that create variable length zero-terminated strings and generally provide this same functionality.

The second problem, the fact that the external code that manipulates the string's data does not update HLA's string length field, is solvable by computing the length of the zero-terminated string upon return from the external code and updating the length field yourself. A convenient way to handle this operation is to write a *wrapper function* that you call from your code. The wrapper function calls the external code and then computes and updates the HLA string length field before returning to the original caller. This saves having to compute the length on each and every invocation of the external code. The HLA Standard Library provides a string length function that efficiently computes the length of a zero-terminated string. You can call this function upon return from the external code and then store the return result into the HLA dynamic length field.

Some external functions may create their own zero-terminated strings rather than store their string data in a buffer you supply. Such functions will probably not allocate storage for the dynamic and maximum length fields that the HLA string format requires. Therefore, you cannot directly use such string data as an HLA string in your assembly code. There are two ways to handle such string data: (1) copy the zero-terminated string to an HLA string and then manipulate the HLA string, or, (2) process the zero-terminated string using functions that directly manipulate such strings. The HLA strings module provides a set of zero-terminated string functions that let you choose either mechanism. The choice of method (1) or (2) depends entirely upon how you intend to use the string data upon return to your HLA code. If you're going to do considerable string manipulation on that string data within your HLA code (and you want to use the full set of HLA string and pattern matching functions on the string data), it makes a lot of sense to first convert the string to the HLA format. On the other hand, if you're going to do very little manipulation, or if the external function expects your code to update the string data in place (so it can refer to a modified version of the original string data at the original address the external code allocates), then it's probably best to manipulate the string data in-place using a set of zero-terminated string functions. If you need to do considerable string manipulation on some data, but the external code expects you to leave the manipulated string in the original buffer it allocates, you can convert the string to an HLA string, do the modification, and then copy the resulting string back into the original buffer; however, all this copying can be expensive, so you should be careful about using this approach.

The HLA Standard Library provides a small handful of important zero-terminated string functions. This set certainly isn't as extensive as the set of functions available for HLA strings, nor is it as extensive as the set of functions available, for example, in the C Standard Library. However, this small set of functions will probably cover 90-95% of the requirements you'll have for processing zero-terminated strings in HLA code. Generally, if you need other functionality, you can obtain it by calling C Standard Library functions from your HLA code or by first converting the string to an HLA string (and then copying the data back to the original buffer, if necessary). The following subsections describe the functions that the HLA Standard Library provides to support zero-terminated strings.

## 39.1 ZStrings Module

To use the zero-terminated string functions in your application, you will need to include one of the following statements at the beginning of your HLA application:

```
#include( "zstrings.hhf" )
or
#include( "stdlib.hhf" )
```

## 39.2 Zstring Functions

```
procedure zstr.len( zstr:zstring ); @returns( "eax" );
```

The single parameter is the address of a zero-terminated string. This function returns the length of that string in the EAX register.

Note that the *zstr.len* function has a single untyped reference parameter. Generally, you'd pass the name of a buffer variable as the parameter to this function. If the address of the zero-terminated string appears in a register, you'll need to use one of the following three invocations to call this function:

```
// Manual invocation- assumes the string pointer is in EBX:

push( ebx );
call zstr.len;
<< length is in EAX >>
.
.
.
zstr.len( [ebx] ); // zlen expects a memory operand
.
.
.
zstr.len( val ebx ); // Tell HLA to use value of ebx.
```

The *zstr.len* function is especially useful for updating the length field of an HLA string you've passed to some external code that generates a zero-terminated string. Consider the following code that updates the length upon return from an external function:

```
// Allocate sufficient storage to hold the string result the external
// code will produce. 1024 was chosen at random for this example, you'll
// have to pick an appropriate value based on the size of the string
// the external procedure in your code produces.

str.alloc( 1024 );
mov( eax, strVar );
.
.
.
externalFunction( strVal ); // externalFunction overwrites strVal data.
zstr.len( strVal ); // Compute the result string's length
```

```

mov( strVar, ebx );           // Get pointer to string data.
if( eax > (type str.strRec [ebx]).MaxStrLen ) then

    // If there was a string overflow, the overflow may
    // have wiped out some important data somewhere, so
    // it may be too late to raise this exception.  However,
    // better late than not notifying the caller at all.
    // Because the buffer overflow may have corrupted the application's
    // data, the application should attempt to terminate as
    // gracefully as possible at this point.

    raise( ex.StringOverflow );

endif;

// Okay, the string didn't overflow the buffer, update the
// HLA string dynamic length field:

mov( eax, (type str.strRec [ebx]).length );

```

HLA high-level calling sequence examples:

```

zstr.len( zstrValue );
mov( eax, zlen );

```

HLA low-level calling sequence examples:

```

pushd( &zstrValue );
call zstr.len;
mov( eax, zlen );
.
.
.
lea( eax, zStrVar );
push( eax );
call zstr.len;
mov( eax, zlen2 );

```

**procedure zstr.zcmp( zsrc1:zstring; zsrc2:zstring ); @returns( "eax" );**

The *zstr.zcmp* function compares two zero-terminated strings and returns the comparison results in the EAX register and in the x86 flags. This comparison function sets the condition code bits so you can use the standard unsigned condition instructions (jump and set instructions) immediately upon return to test for less than, less than or equal, equal, not equal, greater than, or greater than or equal. This function also returns -1 (\$FFFF\_FFFF), zero, or one in EAX to indicate less than, equal, or greater than (respectively). Note that this function compares *zsrc1* to *zsrc2*. Therefore, this function returns -1 if *zsrc1* < *zsrc2*, zero if *zsrc1* = *zsrc2*, and one if *zsrc1* > *zsrc2*.

This function is especially useful for comparing two zero-terminated strings that some external code returns to your HLA program if you don't need to do any further manipulation of the string data. This function is also useful for comparing an HLA string against a zero-terminated string (since HLA strings are zero terminated). Technically, you could use this function to compare two HLA strings (since they are zero-terminated), but the standard HLA string comparison functions are probably more efficient for this purpose.

HLA high-level calling sequence examples:

```

zstr.zcmp( zstr1, zstr2 );
if( @ae ) then // zstr1 >= zstr2
    .
    .
    .
endif;
zstr.zcmp( someZStr, "Hello World" );
mov( eax, cmpResult );

```

HLA low-level calling sequence examples:

```

lea( eax, SomeCharBuffer );
push( eax );
push( zStrVar ); // Note: zstring vars are pointer vars
call zstr.zcmp;
jnae notAE;
    .
    .
    .
notAE:

push( someZStr );
push( HelloWorldStr );
call zstr.zcmp;
mov( eax, cmpResult );

```

#### **procedure zstr.cpy( src:zstring; dest:zstring );**

The *zstr.cpy* function copies one zero-terminated string to another. The destination buffer must be large enough to hold the source string and it is the caller's responsibility to ensure this. The *zstr.cpy* routine has no way to determine the maximum size of the destination buffer, so it cannot check for buffer overflow (this is typical for zero-terminated string functions).

Since HLA strings are zero-terminated, you can use this function to copy an HLA string to a zero-terminated string:

```

// Assumptions: hlaString is the name of an HLA String variable and
// destZStr is the name of an array of characters or byte array.

zstr.cpy( hlaString, destZStr );

```

Of course, you can also use the *zstr.cpy* function to copy one zero-terminated string to another. You'd typically use *zstr.cpy* in this capacity to copy a string returned by one external function to a buffer for use by another external function that expects a zero-terminated string.

#### **procedure zstr.cat( src:zstring; dest:zstring );**

This function concatenates one zero-terminated string to the end of another. The caller must ensure that the destination buffer is large enough to hold the resulting string; the *zstr.cat* function has no way to verify the size of the destination buffer, so it cannot check for buffer overflow (this is typical for zero-terminated string functions).

This string is useful for manipulating zero-terminated strings some external code provides without the overhead of first converting the strings to HLA strings. If you call two external functions that return zero-terminated strings and you need to pass their concatenated result to some other external function that expects a zero-terminated string, and there is no string manipulation in your HLA code, then using *zstr.cat* is more efficient than converting the strings to an HLA string and using the HLA string concation function.



When using this function, don't forget that it's parameters are untyped reference parameters. When passing the address of a buffer variable you may specify the name of the buffer directly. However, when passing a pointer to the buffer, you'll probably need to use the VAL operator to tell HLA to pass the pointer's value rather than the pointer's address. Here are some examples of *zstr.cat* invocations:

```
static
    buffer1    :char[256];
    buffer2    :char[254];
    bufptr1    :zstring;
    bufptr2    :zstring;
    .
    .
    .
    lea( eax, buffer1 );
    mov( eax, bufptr1 );
    lea( eax, buffer2 );
    mov( eax, bufptr2 );
    .
    .
    .
    zstr.cat( bufptr1, bufptr2 );
    zstr.cat( buffer2, edi );
    zstr.cat( bufptr2, esi );
```

You can also use the *zstr.cat* procedure to copy data from an HLA string to a zero-terminated string:

```
static
    hlaStr     :string;
    zs         :char[256];
    zPtr       :zstring;
    .
    .
    .
    lea( eax, zs );
    mov( eax, zPtr );
    .
    .
    .
    zstr.zcat( hlaStr, zPtr );
    zstr.zcat( hlaStr, esi );
```

It really does not make any sense to specify an HLA string variable as the destination operand. *zstr.cat* does not update the HLA string's length field, so if you supply an HLA string as the destination operand, the *zstr.cat* procedure may corrupt the HLA string, forcing you to manually compute the length yourself. If you need to copy a zero-terminated string to an HLA string, use the *zstr.cat* function instead.



# Index

## Numerics

128-bit arithmetic and logical operations  
631

64-Bit arithmetic and logical Operations  
623

## A

a\_adrsToStr (socket module) 761, 778  
 a\_appendFile (blobs module) 54  
 a\_appendFileExtended (blobs module) 55  
 Abstract classes in HOWL 523  
 a\_bToStr (conversions module) 117  
 a\_bufToBlob1 (blobs module) 52  
 a\_bufToBlob2 (blobs module) 52  
 a\_cat (blobs module) 48  
 a\_cat (strings module) 1040  
 a\_catbuf (strings module) 1045  
 a\_catbuf2 (strings module) 1045  
 a\_catbuf3 (strings module) 1045  
 a\_catsub (blobs module) 50  
 a\_catsub (strings module) 1043  
 a\_catz (strings module) 1042  
 accept (socket module) 762  
 AccessDenied (exceptions module) 307  
 AccessViolation (exceptions module) 309  
 Accumulate (timer class) 1134  
 a\_cmdLn (args module) 9  
 a\_columnize2 (strings module) 1022  
 a\_columnize3 (strings module) 1022  
 \_acos (math module) 652  
 acos (math module) 652  
 acos32 (math module) 653  
 acos64 (math module) 653  
 acos80 (math module) 653  
 \_acot (math module) 654  
 acot (math module) 654  
 acot32 (math module) 654  
 acot64 (math module) 654  
 acot80 (math module) 654  
 a\_cpy (blobs module) 42  
 a\_cpy (strings module) 965  
 a\_cpyz (strings module) 965  
 \_acsc (math module) 655  
 acsc (math module) 655  
 acsc32 (math module) 655

acsc64 (math module) 655  
 acsc80 (math module) 655  
 addDays (date module) 287  
 addDays (date/time module) 298  
 addHours (time class module) 1130  
 addHours (time module) 1121  
 addl (math module) 632  
 addMins (time class module) 1130  
 addMins (time module) 1121  
 addMonths (date/time module) 287, 299  
 addq (math routine) 623  
 addSecs (time class module) 1130  
 addSecs (time module) 1121  
 addYears (date/time module) 288, 299  
 a\_delete (strings module) 973  
 a\_delLeadingSpaces (strings module) 975  
 a\_delTrailingSpaces (strings module) 976  
 a\_deTab2 (strings module) 1026  
 a\_deTab3 (strings module) 1027  
 adrsToStr (socket module) 761, 778  
 a\_dToStr (conversions module) 123  
 a\_e32ToStr (conversions module) 236  
 a\_e64ToStr (conversions module) 235  
 a\_e80ToStr (conversions module) 234  
 a\_enTab2 (strings module) 1030  
 a\_enTab3 (strings module) 1030  
 a\_extract (patterns module) 727  
 a\_extractBase (filesystem module) 448  
 a\_extractExt (filesystem module) 450  
 a\_extractFilename (filesystem module) 451  
 a\_extractPath (filesystem module) 452  
 a\_first (strings module) 967  
 afterRow (arrays module) 19  
 a\_get (environment module) 301  
 a\_getField2 (strings module) 1011  
 a\_getField3 (strings module) 1012  
 a\_getFullPathName (filesystem module) 460  
 a\_gets (blobs module) 64  
 a\_gets (file class module) 334  
 a\_gets (file I/O module) 429  
 a\_gets (socket module) 795  
 a\_gets (stdin module) 875  
 a\_h128ToStr (conversions module) 145  
 a\_h16ToStr (conversions module) 136  
 a\_h32ToStr (conversions module) 138  
 a\_h64ToStr (conversions module) 141  
 a\_h80ToStr (conversions module) 143  
 a\_h8ToStr (conversions module) 132

a\_i128ToStr (conversions module) 181  
 a\_i16ToStr (conversions module) 173  
 a\_i32ToStr (conversions module) 176  
 a\_i64ToStr (conversions module) 178  
 a\_i8ToStr (conversions module) 170  
 a\_insert (strings module) 972  
 a\_joinPaths (filesystems module) 453  
 a\_last (strings module) 969  
 alloc (blobs module) 37  
 allocAligned (blobs module) 38  
 allocBlockInHeap (memory module) 677  
 a\_load (blobs module) 53  
 a\_loadExtended (blobs module) 53  
 a\_lower (strings module) 1035  
 a\_lToStr (conversions module) 129  
 andl (math module) 637  
 andq (math module) 628  
 a\_normalize (filesystems module) 455  
 append (lists module) 613  
 appendFile (blobs module) 54  
 append\_index (list module) 613  
 append\_last (list module) 614  
 append\_node (list module) 613  
 appException 489  
 AppException procedure 485  
 Application Framework 485  
 appStart 488  
 AppStart procedure 485  
 appTerminate 489  
 AppTerminate procedure 485  
 a\_qToStr (conversions module) 125  
 a\_r32ToStr (conversions module) 243  
 a\_r64ToStr (conversions module) 243  
 a\_r80ToStr (conversions module) 242  
 arb (patterns module) 728  
 a\_reverse (strings module) 1036  
 arg.a\_cmdLn 9  
 arg.args 12  
 arg.a\_v 11  
 arg.c 10  
 arg.cmdLn 9  
 arg.delete 11  
 arg.destroy 12  
 arg.globalOptions 12  
 arg.localOptions 13  
 arg.v 10  
 args (args module) 12  
 Arithmetic and logical operations (128-bit)  
 631  
 Arithmetic and logical operations (64-bit)  
 623  
 a\_rmvlstWord1 (strings module) 1016  
 a\_rmvlstWord2 (strings module) 1017  
 a\_rmvLastWord1 (strings module) 1018  
 a\_rmvLastWord2 (strings module) 1018  
 a\_rmvTrailingSpaces (strings module)  
 979  
 a\_roman (conversions module) 245  
 Array manipulation 15  
 Array operations 18  
 array.afterRow 19  
 array.beforeRow 19  
 array.cpy 18  
 array.daAlloc 16  
 array.daFree 16  
 array.dArray 15  
 array.element 17  
 array.endreduce 19  
 array.index 17  
 array.IsItDynamic 17  
 array.IsItVar 17  
 array.lookupTable 22  
 array.reduce 19  
 array.reduction 19  
 array.transpose 20  
 ArrayBounds (exceptions module) 308  
 ArrayShapeViolation (exceptions module)  
 308  
 asDword (HLA module) 470  
 \_asec (math module) 656  
 asec (math module) 656  
 asec32 (math module) 656  
 asec64 (math module) 656  
 asec80 (math module) 656  
 \_asin (math module) 651  
 asin (math module) 651  
 asin32 (math module) 651  
 asin64 (math module) 651  
 asin80 (math module) 651  
 AssertionFailed (exceptions module) 306  
 Assignments (blobs module) 42  
 a\_subBlob (blobs module) 44  
 a\_substr (strings module) 966  
 asWord (HLA module) 470  
 Asynchronous remote procedures 745  
 atan (math module) 646

- atan32 (math module) 646
- atan64 (math module) 646
- atan80 (math module) 646
- a\_tbToStr (conversions module) 127
- atof (conversions module) 244
- atoi128 (conversions module) 185
- atoi16 (conversions module) 183
- atoi32 (conversions module) 184
- atoi64 (conversions module) 185
- atoi8 (conversions module) 182
- a\_toNativePath (filesystem module) 460
- a\_toString (date/time module) 291, 299
- a\_toString (time class module) 1131
- a\_toString (time module) 1123
- atou128 (conversions module) 222
- atou16 (conversions module) 220
- atou32 (conversions module) 220
- atou64 (conversions module) 221
- atou8 (conversions module) 219
- a\_toUnixPath (filesystem module) 457
- a\_toWin32Path (filesystem module) 458
- atPos (patterns module) 708
- a\_translate (strings module) 1038
- a\_trim (strings module) 978
- a\_truncate (strings module) 970
- AttemptToDerefNull (exceptions module) 306
- AttemptToFreeNULL (exceptions module) 308
- a\_u128ToStr (conversions module) 217
- a\_u16ToStr (conversions module) 210
- a\_u32ToStr (conversions module) 213
- a\_u64ToStr (conversions module) 215
- a\_u8ToStr (conversions module) 206
- a\_upper (strings module) 1034
- a\_v (args module) 11
- a\_wToStr (conversions module) 120
- B**
- Back tracking 729
- Back tracking support in pattern matching functions 731
- BadFileHandle (exceptions module) 306
- BadObjPtr (exceptions module) 306
- beforeRow (arrays module) 19
- bind (socket module) 762
- Bisynchronous remote procedures 745
- Bit manipulation 25
- Bitmaps (HOWL) 510
- bits.cnt 25
- bits.coalesce 33
- bits.distribute function 32
- bits.extract 32
- bits.merge16 29
- bits.merge32 28
- bits.merge8 29
- bits.nibbles16 31
- bits.nibbles32 30
- bits.nibbles8 31
- bits.reverse16 26
- bits.reverse32 26
- bits.reverse8 27
- bkgColor\_g (in HOWL apps) 488
- Blob accessor functions 39
- Blob assignment functions 42
- Blob comparison functions 45
- Blob concatenation functions 48
- Blob conversion functions 51
- Blob extraction functions 44
- Blob internal representation 36
- Blob scanning functions 45
- Blob variable initialization and allocation 37
- blob.a\_appendFile 54
- blob.a\_appendFileExtended 55
- blob.a\_bufToBlob1 52
- blob.a\_bufToBlob2 52
- blob.a\_cat 48
- blob.a\_catsub 50
- blob.a\_cpy 42
- blob.a\_gets 64
- blob.alloc 37
- blob.allocAligned 38
- blob.a\_load 53
- blob.a\_loadExtended 53
- blob.appendFile 54
- blob.a\_subBlob 44
- blob.bufToBlob2 51
- blob.bufToBlob3 51
- blob.cat2 49
- blob.cat3 49
- blob.catbuf2 50
- blob.catbuf3a 50
- blob.catbuf3b 51
- blob.catbuf4 51
- blob.catsub 49
- blob.catsub4 49

blob.catsub5	49	blob.load	53
blob.chpos	47	blob.maxlen	40
blob.chpos2	47	blob.ne	45
blob.chpos3	47	blob.newln	63
blob.cpy	43	blob.put	64
blob.destroy	39	blob.putb	63
blob.eof	42	blob.putbool	63
blob.eoln	64	blob.putByte	57
blob.eq	45	blob.putc	63
blob.fillb	43	blob.putcset	63
blob.filled	44	blob.putcSize	63
blob.fillw	43	blob.putd	63
blob.free	39	blob.putDword	58
blob.get	64	blob.pute32	64
blob.getBytes	60	blob.pute64	64
blob.getc	64	blob.pute80	64
blob.getDword	61	blob.puth128	63
blob.getf	64	blob.puth16	63
blob.geth128	64	blob.puth16Size	63
blob.geth16	64	blob.puth32	63
blob.geth32	64	blob.puth32Size	63
blob.geth64	64	blob.puth64	63
blob.geth8	64	blob.puth64Size	63
blob.geth80	64	blob.puth8	63
blob.geti128	64	blob.puth80	63
blob.geti16	64	blob.puth80Size	63
blob.geti32	64	blob.puth8Size	63
blob.geti64	64	blob.puti128	63
blob.geti8	64	blob.puti128Size	63
blob.getLword	62	blob.puti16	63
blob.getQword	62	blob.puti16Size	63
blob.gets	64	blob.puti32	63
blob.getTbyte	62	blob.puti32Size	63
blob.getu128	64	blob.puti64	63
blob.getu16	64	blob.puti64Size	63
blob.getu32	64	blob.puti8	63
blob.getu64	64	blob.puti8Size	63
blob.getu8	64	blob.putl	63
blob.getWord	61	blob.putLword	59
blob.index	46	blob.putq	63
blob.index2	46	blob.putQword	58
blob.index3	46	blob.putr32	64
blob.indexStr	46	blob.putr64	64
blob.indexStr2	46	blob.putr80	64
blob.indexStr3	46	blob.puts	63
blob.init	37	blob.putsSize	63
blob.init16	37	blob.puttb	63
blob.length	39	blob.putTbyte	59

blob.putu128 64  
 blob.putu128Size 64  
 blob.putu16 63  
 blob.putu16Size 63  
 blob.putu32 63  
 blob.putu32Size 63  
 blob.putu64 63  
 blob.putu64Size 64  
 blob.putu8 63  
 blob.putu8Size 63  
 blob.putw 63  
 blob.putWord 58  
 blob.rchpos 48  
 blob.rchpos2 48  
 blob.rchpos3 48  
 blob.rcursor 40  
 blob.read 60  
 blob.readAt 60  
 blob.readLn 64  
 blob.realloc 38  
 blob.reset 42  
 blob.rindex 46  
 blob.rindex2 46  
 blob.rindex3 46  
 blob.rindexStr 46  
 blob.rindexStr2 46  
 blob.rindexStr3 46  
 blob.save 55  
 blob.setLength 39  
 blob.setMaxlen 40  
 blob.setrCursor 41  
 blob.setwCursor 41  
 blob.subBlob 44  
 blob.wcursor 41  
 blob.write 55  
 blob.writeAt 57  
 blobRec (blobs module) 36  
 Blobs 35  
 blobs.hhf 35  
 BlockAlreadyFree (exceptions module) 308  
 blockInHeap (memory module) 677  
 BoundInstr (exceptions module) 310  
 Breakpoint (exceptions module) 309  
 brk (strings module) 999  
 brk2 (strings module) 999  
 brk3 (strings module) 1000  
 bSize (conversions module) 92

bToBuf (conversions module) 101  
 bToStr (conversions module) 115  
 bufToBlob2 (blobs module) 51  
 bufToBlob3 (blobs module) 51  
 Buttons (HOWL) 557  
 C  
 c (args module) 10  
 CannotCreateDir (exceptions module) 307  
 CannotFreeMemory (exceptions module) 308  
 CannotRemoveDir (exceptions module) 307  
 CannotRemoveFile (exceptions module) 307  
 CannotRenameFile (exceptions module) 308  
 Case insensitive character matching routines 720  
 case macro 1097  
 cat (strings module) 1041  
 cat (zstrings module) 1140  
 cat2 (blobs module) 49  
 cat2 (strings module) 1041  
 cat3 (blobs module) 49  
 cat3 (strings module) 1041  
 catb (strings module) 1053  
 catbool (strings module) 1047  
 catbuf (strings module) 1046  
 catbuf2 (blobs module) 50  
 catbuf2 (strings module) 1046  
 catbuf3a (blobs module) 50  
 catbuf3a (strings module) 1046  
 catbuf3b (blobs module) 51  
 catbuf3b (strings module) 1046  
 catbuf4 (blobs module) 51  
 catbuf4 (strings module) 1046  
 catc (strings module) 1048  
 catcset (strings module) 1050  
 catcSize (strings module) 1049  
 catd (strings module) 1059  
 cate32 (strings module) 1089  
 cate64 (strings module) 1090  
 cate80 (strings module) 1090  
 cath128 (strings module) 1066  
 cath128Size (strings module) 1067  
 cath16 (strings module) 1057  
 cath16Size (strings module) 1058  
 cath32 (strings module) 1060

cath32Size (strings module) 1060  
 cath64 (strings module) 1062  
 cath64Size (strings module) 1062  
 cath80 (strings module) 1064  
 cath80Size (strings module) 1064  
 cati128 (strings module) 1076  
 cati128Size (strings module) 1077  
 cati16 (strings module) 1071  
 cati16Size (strings module) 1072  
 cati32 (strings module) 1073  
 cati32Size (strings module) 1073  
 cati64 (strings module) 1075  
 cati64Size (strings module) 1075  
 cati8 (strings module) 1068  
 cati8Size (strings module) 1070  
 catl (strings module) 1066  
 catq (strings module) 1061  
 catr32 (strings module) 1091  
 catr64 (strings module) 1092  
 catr80 (strings module) 1093  
 cats (strings module) 1051  
 catsSize (strings module) 1052  
 catsub (blobs module) 49  
 catsub (strings module) 1043  
 catsub4 (blobs module) 49  
 catsub4 (strings module) 1044  
 catsub5 49  
 catsub5 (strings module) 1044  
 cattb (strings module) 1064  
 catu128 (strings module) 1087  
 catu128Size (strings module) 1087  
 catu16 (strings module) 1081  
 catu16Size (strings module) 1082  
 catu32 (strings module) 1083  
 catu32Size (strings module) 1083  
 catu64 (strings module) 1085  
 catu64Size (strings module) 1085  
 catu8 (strings module) 1079  
 catu8Size (strings module) 1080  
 catw (strings module) 1056  
 catz (strings module) 1042  
 cd (filesystem module) 464  
 CDFailed (exceptions module) 307  
 Character classification 65  
 Character concatenation functions 1048  
 Character insertion/removal functions  
     (console module) 78  
 Character matching functions 715  
 Character set concatenation functions  
     1048  
 Character set construction 259  
 Character set searching functions 999  
 Character sets 251  
 Character utilities 65  
 charInStr (strings module) 1010  
 chars.isAlpha 66  
 chars.isAlphaNum 68  
 chars.isASCII 71  
 chars.isCtrl 71  
 chars.isDigit 68  
 chars.isGraphic 70  
 chars.isLower 67  
 chars.isSpace 70  
 chars.isUpper 67  
 chars.isXDigit 69  
 chars.toLower 65  
 chars.toUpper 65  
 charToCset (character sets) 262  
 Check boxes (HOWL) 500  
 Checkable objects (HOWL) 532  
 checkPoint (timer class) 1135  
 chpos (blobs module) 47  
 chpos (strings module) 994  
 chpos2 (blobs module) 47  
 chpos2 (strings module) 994  
 chpos3 (blobs module) 47  
 chpos3 (strings module) 994  
 Circles (HOWL) 505  
 Class traits 955  
 Clickable objects (HOWL) 529  
 Client applications (socket module) 774  
 Client/Server applications 771  
 Client/server communication 774  
 client\_t.close 775  
 client\_t.create 775  
 client\_t.destroy 775  
 Close (file class module) 316  
 Close (file I/O module) 344  
 close (memory-mapped files module) 669  
 Close (socket module) 775  
 close (socket module) 763  
 clrLn (console module) 77  
 clrToBOLN (console module) 77  
 clrToBOScrn (console module) 78  
 clrToEOLN (console module) 77  
 clrToEOScrn (console module) 78



cls (console module) 77  
 cmdLn (args module) 9  
 cnt (bits module) 25  
 coalesce (bits module) 33  
 cocall (coroutines module) 249  
 cofree (coroutines module) 249  
 columnize3 (strings module) 1023  
 columnize4 (strings module) 1023  
 Combo boxes (HOWL) 570  
 Combobox (HOWL) 502  
 Command-line arguments 9  
 Compile-time traits (stl) 956  
 Compiling HOWL applications 492  
 complement (character sets) 270  
 Concatenation functions (blobs module)  
     48  
 connect (socket module) 763  
 Console clearing functions 76  
 Console display control 73  
 Console output colors 81  
 Console scrolling 80  
 console.clrLn 77  
 console.clrToBOLN 77  
 console.clrToBOScrn 78  
 console.clrToEOLN 77  
 console.clrToEOScrn 78  
 console.cls 77  
 console.deleteChar 79  
 console.deleteChars 80  
 console.deleteLine 80  
 console.deleteLines 80  
 console.down 74  
 console.gotorc 73  
 console.gotoxy 73  
 console.home 77  
 console.insertChar 78  
 console.insertChars 78  
 console.insertLine 79  
 console.insertLines 79  
 console.left 75  
 console.ndown 75  
 console.nleft 75  
 console.nright 76  
 console.nup 74  
 console.restoreCursor 76  
 console.right 75  
 console.saveCursor 76  
 console.scrollDown 81  
 console.scrollUp 80  
 console.setAttrs 81  
 console.up 74  
 Container objects (HOWL) 539  
 Containers (HOWL) 541  
 ControlC (exceptions module) 309  
 conv.a\_bToStr 117  
 conv.a\_dToStr 123  
 conv.a\_e32ToStr 236  
 conv.a\_e64ToStr 235  
 conv.a\_e80ToStr 234  
 conv.a\_h128ToStr 145  
 conv.a\_h16ToStr 136  
 conv.a\_h32ToStr 138  
 conv.a\_h64ToStr 141  
 conv.a\_h80ToStr 143  
 conv.a\_h8ToStr 132  
 conv.a\_i128ToStr 181  
 conv.a\_i16ToStr 173  
 conv.a\_i32ToStr 176  
 conv.a\_i64ToStr 178  
 conv.a\_i8ToStr 170  
 conv.a\_lToStr 129  
 conv.a\_qToStr 125  
 conv.a\_r32ToStr 243  
 conv.a\_r64ToStr 243  
 conv.a\_r80ToStr 242  
 conv.a\_roman 245  
 conv.a\_tbToStr 127  
 conv.atof 244  
 conv.atoi128 185  
 conv.atoi16 183  
 conv.atoi32 184  
 conv.atoi64 185  
 conv.atoi8 182  
 conv.atou128 222  
 conv.atou16 220  
 conv.atou32 220  
 conv.atou64 221  
 conv.atou8 219  
 conv.a\_u128ToStr 217  
 conv.a\_u16ToStr 210  
 conv.a\_u32ToStr 213  
 conv.a\_u64ToStr 215  
 conv.a\_u8ToStr 206  
 conv.a\_wToStr 120  
 conv.bSize 92  
 conv.bToBuf 101

conv.bToStr	115	conv._intToBuf128Size	157
conv.dSize	93	conv._intToBuf64Size	157
conv.dToBuf	103	conv.lSize	95
conv.dToStr	122	conv.lToBuf	106
conv.e32ToBuf	231	conv.lToStr	128
conv.e32ToStr	234	conv.qSize	94
conv.e64ToBuf	231	conv.qToBuf	104
conv.e64ToStr	233	conv.qToStr	124
conv.e80ToBuf	230	conv.r32ToBuf	239
conv.e80ToStr	232	conv.r32ToStr	241
conv.getDelimiters	88	conv.r64ToBuf	238
conv.getUnderscores	87	conv.r64ToStr	240
conv.h128Size	100	conv.r80ToBuf	237
conv.h128ToBuf	113	conv.r80ToStr	240
conv.h128ToStr	144	conv.roman	245
conv.h16Size	97	conv.setDelimiters	89
conv.h16ToBuf	109	conv.setUnderscores	85
conv.h16ToStr	134	conv.strToFlt	245
conv.h32Size	97	conv.strToh128	155
conv.h32ToBuf	110	conv.strToh16	152
conv.h32ToStr	137	conv.strToh32	153
conv.h64Size	98	conv.strToh64	154
conv.h64ToBuf	111	conv.strToh8	151
conv.h64ToStr	139	conv.strToi128	191
conv.h80Size	99	conv.strToi16	188
conv.h80ToBuf	112	conv.strToi32	189
conv.h80ToStr	142	conv.strToi64	190
conv.h8Size	96	conv.strToi8	187
conv.h8ToBuf	108	conv.strTou128	228
conv.h8ToStr	130	conv.strTou16	224
conv.hhf	83	conv.strTou32	225
conv.i128Size	160	conv.strTou64	226
conv.i128ToBuf	166	conv.strTou8	223
conv.i128ToStr	179	conv.tbSize	94
conv._i16Size	157	conv.tbToBuf	105
conv.i16Size	158	conv.tbToStr	126
conv.i16ToBuf	163	conv.u128Size	196
conv.i16ToStr	172	conv.u128ToBuf	202
conv.i32Size	159	conv.u128ToStr	216
conv.i32ToBuf	164	conv._u16Size	193
conv.i32ToStr	175	conv.u16Size	194
conv.i64Size	159	conv.u16ToBuf	199
conv.i64ToBuf	165	conv.u16ToStr	208
conv.i64ToStr	177	conv._u32Size	193
conv.i8Size	157	conv.u32Size	195
conv.i8ToBuf	162	conv.u32ToBuf	200
conv.i8ToStr	168	conv.u32ToStr	211
conv._intToBuf128	157	conv.u64Size	195

conv.u64ToBuf 201  
 conv.u64ToStr 214  
 conv.\_u8Size 193  
 conv.u8Size 193  
 conv.u8ToBuf 197  
 conv.u8ToStr 204  
 conv.wSize 93  
 conv.wToBuf 102  
 conv.wToStr 119  
 Conversion format control 84  
 Conversion functions (blobs module) 51  
 ConversionError (exceptions module) 306  
 Conversions 83  
 core (coroutines module) 250  
 Coroutines 247  
 coroutines.hhf 247  
 coroutine\_t.cocall 249  
 coroutine\_t.cofree 249  
 coroutine\_t.create 248  
 \_cos (math module) 642  
 cos (math module) 642  
 cos32 (math module) 642  
 cos64 (math module) 642  
 cos80 (math module) 642  
 \_cot (math module) 647  
 cot (math module) 647  
 cot32 (math module) 647  
 cot64 (math module) 648  
 cot80 (math module) 648  
 Counting the number of set bits in an object  
     25  
 cpy (arrays module) 18  
 cpy (blobs module) 43  
 cpy (character sets) 260  
 cpy (strings module) 965  
 cpy (zstrings module) 1140  
 cpyz (strings module) 966  
 create (coroutines module) 248  
 create (file class module) 314  
 create (list module) 612  
 create (memory-mapped files module) 668  
 create (socket module) 775  
 create (tables module) 1099– 1100  
 create (threads module) 1103  
 create (time class module) 1129  
 create (timer class) 1134  
 createCriticalSection (threads module)  
     1109

createEvent (threads module) 1107  
 createSemaphore (threads module) 1110  
 createTLS (threads module) 1105  
 create\_wBase (HOWL) 525  
 create\_wCheckable (HOWL) 533  
 create\_wClickable (HOWL) 530  
 create\_wFilledFrame (HOWL) 535  
 create\_wSurface 534  
 Creating new list class types 610  
 Critical sections 1108  
 cs.charToCset 262  
 cs.complement 270  
 cs.cpy 260  
 cs.difference 269  
 cs.empty 259  
 cs.eq 257  
 cs.extract 266  
 cs.intersection 268  
 cs.IsEmpty 251  
 cs.member 252  
 cs.ne 258  
 cs.psubset 255  
 cs.psuperset 256  
 cs.rangeChar 263  
 cs.removeChar 272  
 cs.removeStr 275  
 cs.removeStr2 276  
 cs.setunion 268  
 cs.strToCset 265  
 cs.subset 253  
 cs.superset 254  
 cs.unionChar 271  
 cs.unionStr 273  
 cs.unionStr2 274  
 \_csc (math module) 649  
 csc (math module) 649  
 csc32 (math module) 649  
 csc64 (math module) 649  
 csc80 (math module) 649  
 Current system time 1122  
 Cursor position within a pattern 730  
 curTime (date/time module) 1122  
 curTime (time class module) 1130

## D

daAlloc (arrays module) 16  
 daFree (arrays module) 16  
 dArray (arrays module) 15  
 Date functions 279

date.addDays 287  
 date.addMonths 287  
 date.addYears 288  
 date.a\_toString 291  
 date.daterec 279  
 date.dayNumber 285  
 date.dayOfWeek 285  
 date.daysBetween 286  
 date.daysLeft 285  
 date.fromJulian 284, 298  
 date.isLeapYear 281  
 date.isValid 282  
 date.outputFormat 280  
 date.pack 283  
 date.setFormat 290  
 date.setSeparator 290  
 date.subDays 288  
 date.subMonths 288  
 date.subYears 288  
 date.today 289, 299  
 date.toJulian 283  
 date.toString 291  
 date.unpack 283  
 date.utc 289  
 date.validate 282  
 daterec (date/time module) 279  
 DateStarted (timer class) 1134  
 DateStopped (timer class) 1134  
 datetime.hhf 279, 1113  
 dayNumber (date/time module) 285, 298  
 dayOfWeek (date/time module) 285, 298  
 daysBetween (date/time module) 286, 298  
 daysLeft (date/time module) 285, 298  
 Declaring Blob variables 36  
 default macro 1097  
 Deferred evaluation 729  
 Delay functions 681  
 delete (args module) 11  
 delete (filesystem module) 463  
 delete (lists module) 615  
 delete (strings module) 974  
 delete3 (strings module) 974  
 delete4 (strings module) 974  
 deleteChar (console module) 79  
 deleteChars (console module) 80  
 deleteCriticalSection (threads module) 1109  
 deleteEvent (threads module) 1107  
 delete\_first (list module) 616  
 delete\_index (list module) 615  
 delete\_last (list module) 616  
 deleteLine (console module) 80  
 deleteLines (console module) 80  
 delete\_node (list module) 616  
 deleteSemaphore (threads module) 1111  
 delLeadingSpaces (strings module) 975  
 delLeadingSpaces1 (strings module) 976  
 delLeadingSpaces2 (strings module) 976  
 delTrailingSpaces (strings module) 977  
 delTrailingSpaces1 (strings module) 977  
 delTrailingSpaces2 (strings module) 977  
 Deque (stl) 960  
 destroy (args module) 12  
 destroy (blobs module) 39  
 destroy (HOWL) 525  
 destroy (list module) 612  
 destroy (memory-mapped files module) 668  
 destroy (socket module) 775  
 destroy (tables module) 1099– 1100  
 deTab2 (strings module) 1027  
 deTab3a (strings module) 1028  
 deTab3b (strings module) 1028  
 deTab4 (strings module) 1029  
 difference (character sets) 269  
 difference (date/time module) 298  
 difference (time class module) 1130  
 Directory and File manipulation functions 463  
 Directory and file predicates 461  
 dirIn (filesystem module) 467  
 dirInCwd (filesystem module) 467  
 disable (HOWL) 526, 529  
 DiskFullError (exceptions module) 307  
 distribute (bits module) 32  
 DivideError (exceptions module) 310  
 divl (math module) 633  
 divq (math routine) 624  
 down (console module) 74  
 Drag list boxes (HOWL) 570  
 dSize (conversions module) 93  
 dtClass.make\_timeClass 1125  
 dToBuf (conversions module) 103  
 dToStr (conversions module) 122  
 durationToSecs (time module) 1116

## E

- e32ToBuf (conversions module) 231
- e32ToStr (conversions module) 234
- e64ToBuf (conversions module) 231
- e64ToStr (conversions module) 233
- e80ToBuf (conversions module) 230
- e80ToStr (conversions module) 232
- Eager evaluation 729
- Edit boxes (HOWL) 504, 566
- Editors (HOWL) 566
- element (arrays module) 17
- elementsAreObjects\_c (stl) 956– 957, 959
- Ellipses (HOWL) 505
- empty (character sets) 259
- enable (HOWL) 526, 529
- enabled (HOWL) 525
- endMatch (pattern matching module) 707
- EndOfFile (exceptions module) 307
- endreduce (arrays module) 19
- endswitch macro 1097
- endwForm 486, 496
- endwMainMenu 497
- endwRadioSet 512
- endwSubMenu 498
- enTab2 (strings module) 1031
- enTab3a (strings module) 1032
- enTab3b (strings module) 1032
- enTab4 (strings module) 1033
- enterCriticalSection (threads module) 1109
- env.a\_get 301
- env.get 301
- env.hhf 301
- Environment variables 301
- eof (blobs module) 42
- eof (file I/O module) 346
- eoln (blobs module) 64
- eoln (file I/O module) 427
- eoln (stdin module) 873
- eoln2 (stdin module) 873
- eq (blobs module) 45
- eq (character sets) 257
- eq (strings module) 980
- Events (threads module) 1107
- ex.AccessDenied 307
- ex.AccessViolation 309
- ex.ArrayBounds 308
- ex.ArrayShapeViolation 308
- ex.AssertionFailed 306
- ex.AttemptToDerefNULL 306
- ex.AttemptToFreeNULL 308
- ex.BadFileHandle 306
- ex.BadObjPtr 306
- ex.BlockAlreadyFree 308
- ex.BoundInstr 310
- ex.Breakpoint 309
- ex.CannotCreateDir 307
- ex.CannotFreeMemory 308
- ex.CannotRemoveDir 307
- ex.CannotRemoveFile 307
- ex.CannotRenameFile 308
- ex.CDFailed 307
- ex.ControlC 309
- ex.ConversionError 306
- ex.DiskFullError 307
- ex.DivideError 310
- ex.EndOfFile 307
- ex.exceptionMsg 311
- ex.ExecutedAbstract 306
- ex.fDenormal 310
- ex.fDivByZero 310
- ex.FileCloseError 307
- ex.FileNotFound 306
- ex.FileOpenFailure 307
- ex.FileReadError 307
- ex.FileSeekError 307
- ex.FileWriteError 307
- ex.flnexactResult 310
- ex.flInvalidOperation 310
- ex.fOverflow 310
- ex.fStackCheck 310
- ex.fUnderflow 310
- ex.IllegalChar 306
- ex.IllegalInstr 309
- ex.InPageError 309
- ex.IntoInstr 310
- ex.InvalidAlignment 306
- ex.InvalidDate 308
- ex.InvalidDateFormat 308
- ex.InvalidHandle 309
- ex.InvalidTime 309
- ex.InvalidTimeFormat 309
- ex.MemoryAllocationFailure 308
- ex.MemoryFreeFailure 308
- ex.NoMemory 309

ex.PointerNotInHeap 308  
 ex.printStackTrace 311  
 ex.PrivInstr 309  
 ex.SingleStep 309  
 ex.StackOverflow 309  
 ex.StringIndexError 305  
 ex.StringOverflow 305  
 ex.TimeOverflow 309  
 ex.TooManyCmdLnParms 306  
 ex.ValueOutOfRange 305  
 ex.WidthTooBig 306  
 exactlyNChar (patterns module) 717  
 exactlyNCset (patterns module) 713  
 exactlyNiChar (patterns module) 722  
 exactlyNtoMChar (patterns module) 719  
 exactlyNtoMCset (patterns module) 715  
 exactlyNtoMiChar (pattern matching module) 724  
 exceptionMsg (exceptions module) 311  
 Exceptions 303  
 excepts.hhf 303  
 ExecutedAbstract (exceptions module) 306  
 exists (filesys module) 461  
 \_exp (math module) 660  
 exp (math module) 660  
 exp32 (math module) 660  
 exp64 (math module) 660  
 exp80 (math module) 660  
 Extended-precision arithmetic functions 623  
 extract (bits module) 32  
 extract (character sets) 266  
 extract (patterns module) 726  
 extractExt (filesys module) 450  
 extractFilename (filesys module) 451  
 Extraction functions (blobs module) 44  
 extractPath (filesys module) 453

## F

fail (patterns module) 708  
 fastAppend\_c (stl) 956– 957, 959  
 fastElementSwap\_c (stl) 956– 957, 960  
 fastInsert\_c (stl) 956– 957, 959  
 fastPrepend\_c (stl) 956– 957, 959  
 fastRemove\_c (stl) 956– 957, 959  
 fastSearch\_c (stl) 956– 957, 960  
 fastSwap\_c (stl) 956– 957, 960  
 fd\_clr (socket module) 769  
 fDenormal (exceptions module) 310  
 fd\_isset (socket module) 769  
 fDivByZero (exceptions module) 310  
 fd\_set (socket module) 769  
 fd\_zero (socket module) 769  
 fence (patterns module) 708  
 File class 313  
 File I/O module 341  
 File information functions 462  
 File input routines 425  
 File system routines 445  
 file.create 314  
 fileclass.hhf 313  
 FileCloseError (exceptions module) 307  
 fileIn (filesys module) 465  
 fileInCwd (filesys module) 466  
 fileio.a\_gets 429  
 fileio.append 348  
 fileio.Close 344  
 fileio.eof 346  
 fileio.eoln 427  
 fileio.flush 345  
 fileio.get (fileio module) 443  
 fileio.getc 428  
 fileio.getf 442  
 fileio.geth128 441  
 fileio.geth16 438  
 fileio.geth32 439  
 fileio.geth64 440  
 fileio.geth8 437  
 fileio.geti128 433  
 fileio.geti16 430  
 fileio.geti32 431  
 fileio.geti64 432  
 fileio.geti8 430  
 fileio.gets 428  
 fileio.getu128 436  
 fileio.getu16 435  
 fileio.getu32 435  
 fileio.getu64 436  
 fileio.getu8 434  
 fileio.hhf 341  
 fileio.newln 356  
 fileio.Open 341  
 fileio.OpenNew 343  
 fileio.position 349  
 fileio.put 424  
 fileio.putb 365

fileio.putbool 357  
 fileio.putc 358  
 fileio.putcset 361  
 fileio.putcsize 360  
 fileio.putd 373  
 fileio.pute32 416, 943  
 fileio.pute64 417  
 fileio.pute80 418  
 fileio.puth128 385  
 fileio.puth128Size 386  
 fileio.puth16 370  
 fileio.puth16Size 371  
 fileio.puth32 374  
 fileio.puth32Size 375  
 fileio.puth64Size 378  
 fileio.puth8 366  
 fileio.puth80 381  
 fileio.puth80Size 382  
 fileio.puth8Size 367  
 fileio.puti128 399  
 fileio.puti128size 400  
 fileio.puti16Size 393  
 fileio.puti16size 391  
 fileio.puti32 394  
 fileio.puti32Size 395  
 fileio.puti64 397  
 fileio.puti64size 398  
 fileio.puti8 388  
 fileio.putl 384  
 fileio.putq 377  
 fileio.putr32 420  
 fileio.putr64 421  
 fileio.putr80 423  
 fileio.puts 362  
 fileio.putssize 363  
 fileio.puttb 380  
 fileio.putu128 413  
 fileio.putu128size 414  
 fileio.putu16 405  
 fileio.putu16size (file I/O module) 407  
 fileio.putu32 408  
 fileio.putu32size 409  
 fileio.putu64 411  
 fileio.putu64size 412  
 fileio.putu8 402  
 fileio.putu8size 404  
 fileio.putw 369  
 fileio.read 425

fileio.ReadLn 426  
 fileio.rewind 347  
 fileio.rSeek 351  
 fileio.seek 350  
 fileio.size 353  
 fileio.truncate 352  
 fileio.write 354  
 Filename and pathname string functions 445  
 FileNotFound (exceptions module) 306  
 FileOpenFailure 307  
 FileReadError (exceptions module) 307  
 FileSeekError (exceptions module) 307  
 fileys.a\_extractBase 448  
 fileys.a\_extractExt 450  
 fileys.a\_extractFilename 451  
 fileys.a\_extractPath 452  
 fileys.a\_getFullPathName 460  
 fileys.a\_joinPaths 453  
 fileys.a\_normalize 455  
 fileys.a\_toNativePath 460  
 fileys.a\_toUnixPath 457  
 fileys.a\_toWin32Path 458  
 fileys.cd 464  
 fileys.delete 463  
 fileys.dirIn 467  
 fileys.dirInCwd 467  
 fileys.exists 461  
 fileys.extractExt 450  
 fileys.extractFilename 451  
 fileys.extractPath 453  
 fileys.fileIn 465  
 fileys.fileInCwd 466  
 fileys.fileWithSuffix 465  
 fileys.getFullPath 460  
 fileys.gwd 464  
 fileys.hasDriveLetter 445  
 fileys.hasExtension 446  
 fileys.hasPath 448  
 fileys.hasUncName 447  
 fileys.hhf 445  
 fileys.isDir 462  
 fileys.isFile 461  
 fileys.itemInCwd 468  
 fileys.itemWithSuffix 468  
 fileys.joinPaths 454  
 fileys.mkdir 463  
 fileys.normalize1 456

fileys.normalize2 456  
 fileys.rename 464  
 fileys.rmdir 465  
 fileys.size 462  
 fileys.toNativePath1 460  
 fileys.toNativePath2 460  
 fileys.toUnixPath1 457  
 fileys.toUnixPath2 458  
 fileys.toWin32Path1 459  
 fileys.toWin32Path2 459  
 filevar.a\_gets 334  
 filevar.Close 316  
 filevar.get 339  
 filevar.getc 333  
 filevar.getf 339  
 filevar.geth128 339  
 filevar.geth16 338  
 filevar.geth32 338  
 filevar.geth64 338  
 filevar.geth8 337  
 filevar.geti128 336  
 filevar.geti16 335  
 filevar.geti32 335  
 filevar.geti64 335  
 filevar.geti8 334  
 filevar.gets 333  
 filevar.getu16 336  
 filevar.getu32 336  
 filevar.getu64 337  
 filevar.getu8 336  
 filevar.handle 315  
 filevar.newln 318  
 filevar.Open 315  
 filevar.OpenNew 315  
 filevar.put 332  
 filevar.putb 319  
 filevar.putbool 317  
 filevar.putc 318  
 filevar.putcset 318  
 filevar.putcsize 318  
 filevar.putd 321  
 filevar.pute32 329  
 filevar.pute64 329  
 filevar.pute80 330  
 filevar.puth16Size 321  
 filevar.puth64Size 322  
 filevar.puth8Size 320  
 filevar.puti128 326  
 filevar.puti128size 326  
 filevar.puti16 324  
 filevar.puti16size 325  
 filevar.puti32 325  
 filevar.puti32size 325  
 filevar.puti64 325  
 filevar.puti64size 325  
 filevar.putr32 330  
 filevar.putr64 331  
 filevar.puts 319  
 filevar.putssize 319  
 filevar.puttb 322  
 filevar.putu16size 327  
 filevar.putu8 326  
 filevar.putw 320  
 filevar.read 333  
 filevar.ReadLn 333  
 filevar.write 317  
 fileWithSuffix (fileys module) 465  
 FileWriteError (exceptions module) 307  
 fillb (blobs module) 43  
 filld (blobs module) 44  
 fillw (blobs module) 43  
 filteredNodeInList (lists module) 618  
 filteredNodeInListReversed (lists module) 618  
 findInCset (strings module) 1003  
 findInCset2 (strings module) 1003  
 findInCset3 (strings module) 1003  
 flnexactResult (exceptions module) 310  
 flInvalidOperation (exceptions module) 310  
 first (strings module) 968  
 first2 (strings module) 968  
 first3 (strings module) 968  
 firstNChar (patterns module) 718  
 firstNCset (patterns module) 713  
 firstNiChar (patterns module) 722  
 Fixed length hexadecimal numeric to nuff-  
 er conversions 101  
 Fixed size hexadecimal size functions 92  
 Floating-point concatenation functions 1088  
 Floating-point functions 640  
 Floating-point numeric output using scien-  
 tific notation (socket module) 790  
 Floating-point numeric to buffer conver-  
 sions, exponential form 230



flush (file I/O module) 345  
 FlushInput (stdin module) 873  
 Forms (HOWL) 541  
 fOverflow (exceptions module) 310  
 free (blobs module) 39  
 free (memory module) 673  
 freeBlockInHeap (memory module) 678  
 fromJulian (date/time module) 284, 298  
 fromSecs (time class module) 1129  
 fromSecs (time module) 1117  
 fromUnixTime (time module) 1118  
 fromWinFileTime (time module) 1119  
 fStackCheck (exceptions module) 310  
 fUnderflow (exceptions module) 310

## G

ge (strings module) 983  
 get (blobs module) 64  
 get (environment module) 301  
 get (file class module) 339  
 get (fileio module) 443  
 get (socket module) 801  
 get (stdin module) 883  
 getAdrs (socket module) 777  
 getByte (blobs module) 60  
 getc (blobs module) 64  
 getc (file class module) 333  
 getc (file I/O module) 428  
 getc (socket module) 795  
 getc (stdin module) 874  
 getCurrentThreadHandle (threads module) 1105  
 getDelimiters (conversions module) 88  
 getDword (blobs module) 61  
 get\_enabled (HOWL) 526  
 get\_exStyle (HOWL) 528  
 getf (blobs module) 64  
 getf (file class module) 339  
 getf (file I/O module) 442  
 getf (socket module) 800  
 getf (stdin module) 883  
 getField3 (strings module) 1013  
 getField4 (strings module) 1013  
 getFileName (memory-mapped files module) 669  
 getFullPath (filesystem module) 460  
 geth128 (blobs module) 64  
 geth128 (file class module) 339  
 geth128 (file I/O module) 441  
 geth128 (socket module) 800  
 geth128 (stdin module) 877  
 geth16 (blobs module) 64  
 geth16 (file class module) 338  
 geth16 (file I/O module) 438  
 geth16 (socket module) 799  
 geth16 (stdin module) 876  
 geth32 (blobs module) 64  
 geth32 (file class module) 338  
 geth32 (file I/O module) 439  
 geth32 (socket module) 799  
 geth32 (stdin module) 876  
 geth64 (blobs module) 64  
 geth64 (file class module) 338  
 geth64 (file I/O module) 440  
 geth64 (socket module) 800  
 geth64 (stdin module) 877  
 geth8 (blobs module) 64  
 geth8 (file class module) 337  
 geth8 (file I/O module) 437  
 geth8 (socket module) 799  
 geth8 (stdin module) 875  
 geth80 (blobs module) 64  
 get\_handle (HOWL) 526  
 get\_height (HOWL) 528  
 gethostbyaddr (socket module) 769  
 gethostbyname (socket module) 768  
 gethostname (socket module) 768  
 geti128 (blobs module) 64  
 geti128 (file class module) 336  
 geti128 (file I/O module) 433  
 geti128 (socket module) 797  
 geti128 (stdin module) 880  
 geti16 (blobs module) 64  
 geti16 (file class module) 335  
 geti16 (file I/O module) 430  
 geti16 (socket module) 796  
 geti16 (stdin module) 878  
 geti32 (blobs module) 64  
 geti32 (file class module) 335  
 geti32 (file I/O module) 431  
 geti32 (socket module) 796  
 geti32 (stdin module) 879  
 geti64 (blobs module) 64  
 geti64 (file class module) 335  
 geti64 (file I/O module) 432  
 geti64 (socket module) 797  
 geti64 (stdin module) 879

geti8 (blobs module) 64  
 geti8 (file class module) 334  
 geti8 (file I/O module) 430  
 geti8 (socket module) 796  
 geti8 (stdin module) 878  
 getLword (blobs module) 62  
 getMalloc (memory-mapped files module) 670  
 getNode (tables module) 1099, 1102  
 get\_objectID (HOWL) 526  
 get\_onHeap (HOWL) 526  
 getOpen (memory-mapped files module) 669  
 get\_parentForm (HOWL) 526  
 get\_parentHandle (HOWL) 526, 528  
 getpeername (socket module) 769  
 getPort (socket class module) 778  
 getPos (patterns module) 708  
 getQword (blobs module) 62  
 getref (memory module) 676, 679  
 gets (blobs module) 64  
 gets (file class module) 334  
 gets (file I/O module) 428  
 gets (socket module) 795  
 gets (stdin module) 874  
 getsockname (socket module) 769  
 get\_style (HOWL) 528  
 getTbyte (blobs module) 62  
 getTimeout (socket class module) 777  
 getTLS (threads module) 1106  
 getu128 (blobs module) 64  
 getu128 (file class module) 337  
 getu128 (file I/O module) 436  
 getu128 (socket module) 798  
 getu128 (stdin module) 882  
 getu16 (blobs module) 64  
 getu16 (file class module) 336  
 getu16 (file I/O module) 435  
 getu16 (socket module) 798  
 getu16 (stdin module) 881  
 getu32 (blobs module) 64  
 getu32 (file class module) 336  
 getu32 (file I/O module) 435  
 getu32 (socket module) 798  
 getu32 (stdin module) 881  
 getu64 (blobs module) 64  
 getu64 (file class module) 337  
 getu64 (file I/O module) 436

getu64 (socket module) 798  
 getu64 (stdin module) 882  
 getu8 (blobs module) 64  
 getu8 (file class module) 336  
 getu8 (file I/O module) 434  
 getu8 (socket module) 797  
 getu8 (stdin module) 880  
 get\_visible (HOWL) 526  
 getWhiteSpace (patterns module) 727  
 get\_width (HOWL) 528  
 getWord (blobs module) 61  
 getWordDelims (patterns module) 726  
 get\_x (HOWL) 528  
 get\_y (HOWL) 528  
 globalOptions (args module) 12  
 gotorc (console module) 73  
 gotoxy (console module) 73  
 Graphic objects (HOWL) 548  
 gt (strings module) 982  
 gwd (filesystem module) 464

## H

h128Size (conversions module) 100  
 h128ToBuf (conversions module) 113  
 h128ToStr (conversions module) 144  
 h16Size (conversions module) 97  
 h16ToBuf (conversions module) 109  
 h16ToStr (conversions module) 134  
 h32Size (conversions module) 97  
 h32ToBuf (conversions module) 110  
 h32ToStr (conversions module) 137  
 h64Size (conversions module) 98  
 h64ToBuf (conversions module) 111  
 h64ToStr (conversions module) 139  
 h80Size (conversions module) 99  
 h80ToBuf (conversions module) 112  
 h80ToStr 142  
 h8Size (conversions module) 96  
 h8ToBuf (conversions module) 108  
 h8ToStr (conversions module) 130  
 handle (file class module) 315  
 handle (HOWL) 524  
 handle (stdin module) 871  
 handle (stdio module) 885  
 hasDriveLetter (filesystem module) 445  
 hasExtension (filesystem module) 446  
 hasPath (filesystem module) 448  
 hasUncName (filesystem module) 447  
 Hexadecimal concatenation functions

- 1053*
- Hexadecimal conversions *91*
- Hexadecimal input from a socket *799*
- Hexadecimal numeric size functions *92*
- Hexadecimal numeric to buffer conversions *101*
- Hexadecimal numeric to string conversions *115*
- Hexadecimal size functions *96*
- hide (HOWL) *526, 529*
- High-level language statements *1097*
- HLA macros and constants *469*
- HLA Object Windows Library *485*
- HLA string data type *961*
- hla.asDword *470*
- hla.asWord *470*
- hla.hhf *469*
- hla.IsHex *469*
- hla.IsInt *469*
- hla.IsNumber *469*
- hla.IsNumeric *470*
- hla.IsOrdinal *470*
- hla.IsReal *469*
- hla.IsUns *469*
- \_hla.make\_listClass *610*
- hll.cswitch *1098*
- hll.usebubblesort *1098*
- home (console module) *77*
- HOWL *485*
- HOWL Declarative Language *495*
- HOWL object hierarchy *522*
- HOWL Run-time Library *521*
- HowlMainApp *485*
- I**
- i128Size (conversions module) *160*
- i128TBuf (conversions module) *166*
- i128ToStr (conversions module) *179*
- \_i16Size (conversions module) *157*
- i16Size (conversions module) *158*
- i16ToBuf (conversions module) *163*
- i16ToStr (conversions module) *172*
- i32Size (conversions module) *159*
- i32ToBuf (conversions module) *164*
- i32ToStr (conversions module) *175*
- i64Size (conversions module) *159*
- i64ToBuf (conversions module) *165*
- i64ToStr (conversions module) *177*
- i8Size (conversions module) *157*
- i8ToBuf (conversions module) *162*
- i8ToStr (conversions module) *168*
- ichpos (strings module) *995*
- ichpos2 (strings module) *995*
- ichpos3 (strings module) *995*
- Icons (HOWL) *591*
- idivl (math module) *633*
- idivq (math module) *625*
- ieq (strings module) *983*
- ige (strings module) *985*
- igt (strings module) *985*
- iindex (strings module) *989*
- iindex3 (strings module) *990*
- ile (strings module) *985*
- IllegalChar (exceptions module) *306*
- IllegalInstr (exceptions module) *309*
- ilt (strings module) *984*
- imodl (math module) *635*
- imodq (math module) *626*
- imull (math module) *636*
- imulq (math module) *627*
- index (arrays module) *17*
- index (blobs module) *46*
- index (list module) *617*
- index (strings module) *988*
- index2 (blobs module) *46*
- index2 (strings module) *988*
- index3 (blobs module) *46*
- index3 (strings module) *988*
- indexStr *46*
- indexStr2 (blobs module) *46*
- indexStr3 (blobs module) *46*
- ine (strings module) *984*
- init (blobs module) *37*
- init (strings module) *963*
- init16 (blobs module) *37*
- Initializing and allocating blob variables *37*
- InPageError (exceptions module) *309*
- insert (lists module) *614*
- insert (strings module) *972*
- insert3 (strings module) *972*
- insert4 (strings module) *973*
- insertChar (console module) *78*
- insertChars (console module) *78*
- insert\_first (list module) *615*
- insert\_index (list module) *614*
- insertLine (console module) *79*

insertLines (console module) 79  
 insert\_node (list module) 615  
 intersection (character sets) 268  
 IntoInstr (exceptions module) 310  
 \_intToBuf128 (conversions module) 157  
 \_intToBuf128Size (conversions module) 157  
 \_intToBuf32 (conversions module) 157  
 \_intToBuf32Size (conversions module) 157  
 \_intToBuf64Size (conversions module) 157  
 InvalidArgument (exceptions module) 306  
 InvalidDate (exceptions module) 308  
 InvalidDateFormat (exceptions module) 308  
 InvalidHandle (exceptions module) 309  
 InvalidTime (exceptions module) 309  
 InvalidTimeFormat (exceptions module) 309  
 irchpos (strings module) 997  
 irchpos2 (strings module) 998  
 irchpos3 (strings module) 998  
 irindex (strings module) 992  
 irindex2 (strings module) 992  
 irindex3 (strings module) 993  
 isAlloc field in stl classes 960  
 isAlpha (chars module) 66  
 isAlphaNum (chars module) 68  
 IsArray\_c (stl) 956– 957  
 isArray\_c (stl) 958  
 isASCII (chars module) 71  
 isContainer\_c (stl) 956– 957  
 isCtrl (chars module) 71  
 isDeque\_c (stl) 956– 958  
 isDigit (chars module) 68  
 isDir (filesystem module) 462  
 IsEmpty (character sets) 251  
 isFile (filesystem module) 461  
 isGraphic (chars module) 70  
 IsHex (HLA module) 469  
 isInHeap (memory module) 674  
 IsInt (HLA module) 469  
 IsItDynamic (arrays module) 17  
 IsItVar (arrays module) 17  
 isLeapYear (date/time module) 281, 298  
 isList\_c (stl) 956– 958  
 isLower (chars module) 67  
 IsNumber (HLA module) 469  
 IsNumeric (HLA module) 470  
 IsOrdinal (HLA module) 470  
 isRandomAccess\_c (stl) 956– 958  
 IsReal (HLA module) 469  
 isSpace (chars module) 70  
 isSTL\_c 955  
 isTable\_c (stl) 956– 958  
 IsUns (HLA module) 469  
 isUpper (chars module) 67  
 isValid (date/time module) 282, 298  
 isValid (time class module) 1129  
 \_isValid (time module) 1115  
 isValid (time module) 1115  
 isValue (date/time module) 1115  
 isVector\_c (stl) 956– 957  
 isXDigit (chars module) 69  
 item (tables module) 1099  
 itemInCwd (filesystem module) 468  
 itemWithSuffix (filesystem module) 468  
  
 J  
  
 L  
 l\_arb (patterns module) 729  
 last (strings module) 969  
 last2 (strings module) 969  
 last3 (strings module) 970  
 Lazy/deferred evaluation 729  
 le (strings module) 982  
 leaveCriticalSection (threads module) 1110  
 left (console module) 75  
 Left-text check boxes 500  
 Left-text radio buttons (HOWL) 512  
 Left-text radio set buttons (HOWL) 513  
 len (zstrings module) 1138  
 length (blobs module) 39  
 length (strings module) 964  
 l\_ExactlyNtoMChar (patterns module) 720  
 l\_ExactlyNtoMCset (patterns module) 715  
 l\_ExactlyNtoMiChar (patterns module) 724  
 Linux 605  
 linux.hhf 605  
 List (stl) 960

- List boxes (HOWL) 508, 570
- List constructor and destructor 612
- List procedures, methods, and iterators 611
- list.append\_index 613
- list.append\_last 614
- list.append\_node 613
- list.create 612
- list.delete\_first 616
- list.delete\_index 615
- list.delete\_last 616
- list.delete\_node 616
- list.destroy 612
- list.index 617
- list.insert\_first 615
- list.insert\_index 614
- list.insert\_node 615
- list.numNodes 613
- listen (socket module) 763
- Lists 607
- lists.hhf 607
- list\_t (lists module) 607
- list\_t.append 613
- list\_t.delete 615
- list\_t.filteredNodeInList 618
- list\_t.filteredNodeInListReversed 618
- list\_t.insert 614
- list\_t.nodeInList 617
- list\_t.nodeInListReversed 617
- list\_t.reverse 619
- list\_t.search 620
- list\_t.sort 620
- list\_t.xchgNodes 620
- \_ln (math module) 664
- ln (math module) 664
- ln32 (math module) 664
- ln64 (math module) 664
- ln80 (math module) 664
- l\_NorLessChar (patterns module) 718
- l\_NorLessCset (patterns module) 713
- l\_NorLessiChar (patterns module) 723
- l\_NorMoreChar (patterns module) 719
- l\_NorMoreCset (patterns module) 714
- l\_NorMoreiChar (patterns module) 723
- l\_NtoMChar (patterns module) 719
- l\_NtoMCset (patterns module) 714
- l\_NtoMiChar (patterns module) 724
- load (blobs module) 53
- localOptions (args module) 13
- \_log (math module) 663
- log (math module) 663
- log32 (math module) 663
- log64 (math module) 663
- log80 (math module) 663
- Logarithmic arithmetic operations 640
- l\_OneOrMoreChar (patterns module) 717
- l\_OneOrMoreCset (patterns module) 712
- l\_OneOrMoreiChar (patterns module) 722
- lookup (tables module) 1099, 1101
- Lookup tables 22
- lookupTable (arrays module) 22
- lower (strings module) 1036
- lower1 (strings module) 1036
- lower2 (strings module) 1036
- lSize (conversions module) 95
- lt (strings module) 981
- lToBuf (conversions module) 106
- lToStr (conversions module) 128
- l\_ZeroOrMoreChar (patterns module) 717
- l\_ZeroOrMoreCset (patterns module) 712
- l\_ZeroOrMoreiChar (patterns module) 721
- l\_ZeroOrOneChar (patterns module) 716
- l\_ZeroOrOneCset (patterns module) 711
- l\_ZeroOrOneiChar (patterns module) 721
- M
- make\_timeClass (time class module) 1125
- Mashalling parameters 745
- match (pattern matching module) 707
- Matching an arbitrary sequence of characters 728
- matchiStr (patterns module) 725
- matchiWord (patterns module) 726
- matchStr (patterns module) 724
- matchToiStr (patterns module) 725
- matchToStr (patterns module) 725
- matchWord (patterns module) 725
- Math functions 623
- math.\_acos 652
- math.acos 652
- math.acos32 653
- math.acos64 653
- math.acos80 653
- math.\_acot 654
- math.acot 654
- math.acot32 654

math.acot64	654	math.imodl	635
math.acot80	654	math.imodq	626
math._acsc	655	math.imull	636
math.acsc	655	math.imulq	627
math.acsc32	655	math._ln	664
math.acsc64	655	math.ln	664
math.acsc80	655	math.ln32	664
math.addl	632	math.ln64	664
math.addq	623	math.ln80	664
math.andl	637	math._log	663
math.andq	628	math.log	663
math._asec	656	math.log32	663
math.asec	656	math.log64	663
math.asec32	656	math.log80	663
math.asec64	656	math.modl	634
math.asec80	656	math.modq	626
math._asin	651	math.mull	635
math.asin	651	math.mulq	627
math.asin32	651	math.negl	636
math.asin64	651	math.negq	628
math.asin80	651	math.notl	639
math._atan	646	math.notq	630
math.atan	646	math.orl	638
math.atan32	646	math.orq	629
math.atan64	646	math._sec	650
math.atan80	646	math.sec32	650
math._cos	642	math.sec64	650
math.cos	642	math.sec80	650
math.cos32	642	math.shll	639
math.cos64	642	math.shlq	630
math.cos80	642	math.shrl	640
math._cot	647	math.shrq	631
math.cot32	647	math._sin	641
math.cot64	648	math.sin	641
math.cot80	648	math.sin32	641
math._csc	649	math.sin64	641
math.csc32	649	math.sin80	641
math.csc64	649	math._sincos	645
math.csc80	649	math.sincos	645
math.divl	633	math.sincos32	645
math._exp	660	math.sincos64	645
math.exp	660	math.sincos80	645
math.exp32	660	math.subl	632
math.exp64	660	math.subq	624
math.exp80	660	math._tan	644
math.hhf	623	math.tan	644
math.idivl	633	math.tan32	644
math.idivq	625	math.tan64	644

math.tan80 644  
 math.TenToX 659  
 math.\_tenToX 659  
 math.tenToX32 659  
 math.tenToX64 659  
 math.tenToX80 659  
 math.TwoToX 658  
 math.\_twoToX 658  
 math.twoToX32 658  
 math.twoToX64 658  
 math.twoToX80 658  
 math.xorl 638  
 math.xorq 629  
 math.\_YtoX 661  
 math.ytoX 661  
 math.YtoX32 661  
 math.YtoX64 661  
 math.YtoX80 661  
 maxlen (blobs module) 40  
 mem.alloc 671  
 mem.alloc1 (memory module) 671  
 mem.alloc2 (memory module) 672  
 mem.allocBlockInHeap 677  
 mem.blockInHeap (memory module) 677  
 mem.free 673  
 mem.freeBlockInHeap 678  
 mem.getref 676  
 mem.isInHeap 674  
 mem.isInHeap (memory module) 674  
 mem.newref 676  
 mem.realloc 673  
 mem.realloc1 673  
 mem.realloc2 673  
 mem.size 675  
 mem.stat 675  
 mem.talloc 674  
 mem.zalloc 672  
 member (character sets) 252  
 Memory management 671  
 memory.hhf 671  
 MemoryAllocationFailure (exceptions module) 308  
 MemoryFreeFailure (exceptions module) 308  
 Memory-mapped I/O 667  
 Menu separator items 498  
 Menu widgets (HOWL) 497  
 Menus (HOWL) 541  
 merge32, merge16, merge8 (bits module) 28  
 mkdir (filesystem module) 463  
 mmap.hhf 667  
 mmap\_t.close 669  
 mmap\_t.create 668  
 mmap\_t.destroy 668  
 mmap\_t.getFileName 669  
 mmap\_t.getMalloc 670  
 mmap\_t.getOpen 669  
 mmap\_t.open 668  
 mmap\_t.openNew 668  
 modl (math module) 634  
 modq (math module) 626  
 move (HOWL) 528  
 mSleep (os module) 682  
 msStarted (timer class) 1134  
 msStopped (timer class) 1134  
 mull (math module) 635  
 mulq (math module) 627  
 N  
 \_name (HOWL) 525  
 ndown (console module) 75  
 ne (blobs module) 45  
 ne (character sets) 258  
 ne (strings module) 981  
 negl (math module) 636  
 negq (math module) 628  
 newln (blobs module) 63  
 newln (file class module) 318  
 newln (file I/O module) 356  
 newln (socket module) 779  
 newln (stderr module) 805  
 newln (stdout module) 887  
 newref (memory module) 676  
 nextWidget (HOWL) 525  
 nibbles32, nibbles16, nibbles8 (bits module) 30  
 nleft (console module) 75  
 nodeInList (lists module) 617  
 nodeInListReversed (lists module) 617  
 nodePtr\_t (lists module) 607  
 NoMemory (exceptions module) 309  
 norLessChar (patterns module) 718  
 norLessCset (patterns module) 713  
 norLessiChar (patterns module) 722  
 normalize1 (filesystem module) 456  
 normalize2 (filesystem module) 456

norMoreChar (patterns module) 718  
 norMoreCset (patterns module) 714  
 norMoreiChar (patterns module) 723  
 notl (math module) 639  
 notq (math routine) 630  
 nright (console module) 76  
 ntoMChar (patterns module) 719, 723  
 ntoMCset (patterns module) 714  
 numNodes (list module) 613  
 nup (console module) 74

## O

objectID (HOWL) 525  
 onClick (HOWL) 530  
 onClose (HOWL) 529  
 onCreate (HOWL) 529  
 onDbClick (HOWL) 530  
 oneChar (patterns module) 716  
 oneCset (patterns module) 710  
 oneiChar (patterns module) 720  
 oneOrMoreChar (patterns module) 717  
 oneOrMoreCset (patterns module) 712  
 oneOrMoreiChar (patterns module) 721  
 oneOrMorePat (patterns module) 710  
 oneOrMoreWS (patterns module) 728  
 onePat (patterns module) 709  
 One-shot timers (HOWL) 601  
 onHeap (HOWL) 525  
 onPaint (HOWL) 534  
 Open (file class module) 315  
 Open (file I/O module) 341  
 open (memory-mapped files module) 668  
 OpenNew (file class module) 315  
 OpenNew (file I/O module) 343  
 openNew (memory-mapped files module) 668  
 orl (math module) 638  
 orq (math routine) 629  
 OS functions 681  
 os.hhf 681  
 os.mSleep 682  
 os.sleep 682  
 os.system 681  
 OutputFormat (date/time module) 280

## P

pack (date/time module) 283  
 pack (time module) 1116  
 parentForm (HOWL) 525  
 parentHandle (HOWL) 525

Passing byte parameters on the stack 3  
 Passing dword parameters on the stack 5  
 Passing lword parameters on the stack 7  
 Passing parameters by reference and by value 2  
 Passing parameters to Standard Library routines 1  
 Passing qword parameters on the stack 6  
 Passing tbyte parameters on the stack 7  
 Passing word parameters on the stack 5  
 Passwords (HOWL) 508  
 pat.a\_extract 727  
 pat.arb 728  
 pat.atPos 708  
 pat.endMatch 707  
 pat.exactlyNChar 717  
 pat.exactlyNCset 713  
 pat.exactlyNiChar 722  
 pat.exactlyNtoMChar 719  
 pat.exactlyNtoMCset 715  
 pat.exactlyNtoMiChar 724  
 pat.extract 726  
 pat.fail 708  
 pat.FailRec object 732  
 pat.fence 708  
 pat.firstNChar 718  
 pat.firstNCset 713  
 pat.firstNiChar 722  
 pat.getPos 708  
 pat.getWhiteSpace 727  
 pat.getWordDelims 726  
 pat.l\_arb 729  
 pat.l\_ExactlyNtoMChar 720  
 pat.l\_ExactlyNtoMCset 715  
 pat.l\_ExactlyNtoMiChar 724  
 pat.l\_NorLessChar 718  
 pat.l\_NorLessCset 713  
 pat.l\_NorLessiChar 723  
 pat.l\_NorMoreChar 719  
 pat.l\_NorMoreCset 714  
 pat.l\_NorMoreiChar 723  
 pat.l\_NtoMChar 719  
 pat.l\_NtoMCset 714  
 pat.l\_NtoMiChar 724  
 pat.l\_OneOrMoreChar 717  
 pat.l\_OneOrMoreCset 712  
 pat.l\_OneOrMoreiChar 722  
 pat.l\_ZeroOrMoreChar 717



pat.l_ZeroOrMoreCset	712	pat.zeroOrMoreCset	712
pat.l_ZeroOrMoreiChar	721	pat.zeroOrMoreiChar	721
pat.l_ZeroOrOneChar	716	pat.zeroOrMorePat	710
pat.l_ZeroOrOneCset	711	pat.zeroOrMoreWS	727
pat.l_ZeroOrOneiChar	721	pat.zeroOrOneChar	716
pat.match	707	pat.zeroOrOneCset	711
pat.matchiStr	725	pat.zeroOrOneiChar	720
pat.matchiWord	726	pat.zeroOrOnePat	710
pat.matchStr	724	Pattern matching	683
pat.matchToiStr	725	peekc (stdin module)	874
pat.matchToStr	725	peekChar (patterns module)	715
pat.matchWord	725	peekCset (patterns module)	710
pat.norLessChar	718	peekiChar (patterns module)	720
pat.norLessCset	713	peekWS (patterns module)	728
pat.norLessiChar	722	peekWSorEOS (patterns module)	728
pat.norMoreChar	718	Periodic timers (HOWL)	601
pat.norMoreCset	714	Pie wedges (HOWL)	509
pat.norMoreiChar	723	PointerNotInHeap (exceptions module)	308
pat.ntoMChar	719		
pat.ntoMCset	714	Polygons (HOWL)	509
pat.ntoMiChar	723	position (patterns module)	707
pat.oneChar	716	Predicates (chars module)	66
pat.oneCset	710	prefix (strings module)	986
pat.oneiChar	720	prefix3 (strings module)	987
pat.oneOrMoreChar	717	printExceptionError (exceptions module)	311
pat.oneOrMoreCset	712		
pat.oneOrMoreiChar	721	PrivInstr (exceptions module)	309
pat.oneOrMorePat	708, 710	processMessage (HOWL)	526
pat.oneOrMoreWS	728	Progress bars (HOWL)	511, 576
pat.onePat pattern matching function	709	Proxy function (remote procedure call)	745
pat.peekChar	715		
pat.peekCset	710	Proxy functions (remote procedure calls)	750
pat.peekiChar	720		
pat.peekWS	728	psubset (character sets)	255
pat.peekWSorEOS	728	psuperset (character sets)	256
pat.position	707	Push buttons (HOWL)	511
pat.setWhiteSpace	727	put (blobs module)	64
pat.setWordDelims	726	put (file class module)	332
pat.skip	708	put (file I/O module)	424
pat._success_macro	732	put (socket module)	793
pat.upToChar	716	put (stderr module)	869
pat.upToCset	711	put (stdout module)	951
pat.upToiChar	720	put (strings module)	1095
pat.upToiStr	725	putb (blobs module)	63
pat.upToStr	725	putb (file class module)	319
pat.WSorEOS	728	putb (file I/O module)	365
pat.WSthenEOS	728	putb (socket module)	781
pat.zeroOrMoreChar	716	putb (stderr module)	813

putb (stdout module)	895
putbool (blobs module)	63
putbool (file class module)	317
putbool (file I/O module)	357
putbool (socket module)	779
putbool (stderr module)	805
putbool (stdout module)	887
putByte (blobs module)	57
putc (blobs module)	63
putc (file class module)	318
putc (file I/O module)	358
putc (socket module)	779
putc (stderr module)	806
putc (stdout module)	888
putcset (blobs module)	63
putcset (file class module)	318
putcset (file I/O module)	361
putcset (socket module)	780
putcset (stderr module)	809
putcset (stdout module)	891
putcSize (blobs module)	63
putcsize (file class module)	318
putcsize (file I/O module)	360
putcsize (socket module)	780
putcSize (stderr module)	808
putcSize (stdout module)	890
putd (blobs module)	63
putd (file class module)	321
putd (file I/O module)	373
putd (socket module)	783
putd (stderr module)	820
putd (stdout module)	902
putDword (blobs module)	58
pute32 (blobs module)	64
pute32 (file class module)	329
pute32 (file I/O module)	416
pute32 (socket module)	790
pute32 (stderr module)	861
pute32 (stdout module)	944
pute64 (blobs module)	64
pute64 (file class module)	329
pute64 (file I/O module)	417
pute64 (socket module)	791
pute64 (stderr module)	862
pute64 (stdout module)	944
pute80 (blobs module)	64
pute80 (file class module)	330
pute80 (file I/O module)	418
pute80 (socket module)	791
pute80 (stderr module)	863
pute80 (stdout module)	945
puth128 (blobs module)	63
puth128 (file class module)	323
puth128 (file I/O module)	385
puth128 (socket module)	785
puth128 (stderr module)	831
puth128 (stdout module)	914
puth128Size (file class module)	323
puth128Size (file I/O module)	386
puth128Size (socket module)	785
puth128Size (stderr module)	832
puth128Size (stdout module)	915
puth16 (blobs module)	63
puth16 (file class module)	320
puth16 (fileio module)	370
puth16 (socket module)	782
puth16 (stderr module)	818
puth16 (stdout module)	900
puth16Size (blobs module)	63
puth16Size (file I/O module)	371
puth16Size (fileclass module)	321
puth16Size (socket module)	782
puth16Size (stderr module)	819
puth16Size (stdout module)	901
puth32 (blobs module)	63
puth32 (file class module)	321
puth32 (file I/O module)	374
puth32 (socket module)	783
puth32 (stderr module)	821
puth32 (stdout module)	903
puth32Size (blobs module)	63
puth32Size (file class module)	321
puth32Size (file I/O module)	375
puth32Size (socket module)	783
puth32Size (stderr module)	822
puth32Size (stdout module)	904
puth64 (blobs module)	63
puth64 (file class module)	322
puth64 (file I/O module)	377
puth64 (socket module)	784
puth64 (stderr module)	824
puth64 (stdout module)	906
puth64Size (blobs module)	63
puth64Size (file class module)	322
puth64Size (file I/O module)	378
puth64Size (socket module)	784

puth64Size (stderr module) 825  
 puth64Size (stdout module) 907  
 puth8 (blobs module) 63  
 puth8 (file class module) 320  
 puth8 (fileio module) 366  
 puth8 (socket module) 781  
 puth8 (stdout module) 896  
 puth80 (blobs module) 63  
 puth80 (file class module) 322  
 puth80 (file I/O module) 381  
 puth80 (socket module) 784  
 puth80 (stderr module) 827  
 puth80 (stdout module) 910  
 puth80Size (blobs module) 63  
 puth80Size (file class module) 323  
 puth80Size (file I/O module) 382  
 puth80Size (socket module) 784  
 puth80Size (stderr module) 828  
 puth80Size (stdout module) 911  
 puth8Size (blobs module) 63  
 puth8Size (file class module) 320  
 puth8Size (file I/O module) 367  
 puth8Size (socket module) 782  
 puth8Size (stderr module) 815  
 puth8Size (stdout module) 897  
 puti128 (blobs module) 63  
 puti128 (file class module) 326  
 puti128 (file I/O module) 399  
 puti128 (socket module) 787  
 puti128 (stderr module) 845  
 puti128 (stdout module) 928  
 puti128Size (blobs module) 63  
 puti128size (file class module) 326  
 puti128size (file I/O module) 400  
 puti128Size (socket module) 788  
 puti128Size (stderr module) 845  
 puti128Size (stdout module) 928  
 puti16 (blobs module) 63  
 puti16 (file class module) 324  
 puti16 (file I/O module) 391  
 puti16 (socket module) 786  
 puti16 (stderr module) 837  
 puti16 (stdout module) 920  
 puti16Size (blobs module) 63  
 puti16size (file class module) 325  
 puti16size (file I/O module) 393  
 puti16size (socket module) 786  
 puti16Size (stderr module) 838  
 puti16Size (stdout module) 921  
 puti32 (blobs module) 63  
 puti32 (file class module) 325  
 puti32 (file I/O module) 394  
 puti32 (socket module) 786  
 puti32 (stderr module) 840  
 puti32 (stdout module) 923  
 puti32Size (blobs module) 63  
 puti32size (file class module) 325  
 puti32size (file I/O module) 395  
 puti32Size (socket module) 787  
 puti32Size (stderr module) 841  
 puti32Size (stdout module) 924  
 puti64 (blobs module) 63  
 puti64 (file class module) 325  
 puti64 (file I/O module) 397  
 puti64 (socket module) 787  
 puti64 (stderr module) 842  
 puti64 (stdout module) 925  
 puti64Size (blobs module) 63  
 puti64size (file class module) 326  
 puti64size (file I/O module) 398  
 puti64Size (socket module) 787  
 puti64Size (stderr module) 843  
 puti64Size (stdout module) 926  
 puti8 (blobs module) 63  
 puti8 (file class module) 324  
 puti8 (file I/O module) 388  
 puti8 (socket module) 786  
 puti8 (stderr module) 834  
 puti8 (stdout module) 917  
 puti8Size (blobs module) 63  
 puti8size (file class module) 324  
 puti8size (file I/O module) 390  
 puti8Size (socket module) 786  
 puti8Size (stderr module) 836  
 puti8Size (stdout module) 919  
 putl (blobs module) 63  
 putl (file class module) 323  
 putl (file I/O module) 384  
 putl (socket module) 785  
 putl (stderr module) 830  
 putl (stdout module) 913  
 putLword (blobs module) 59  
 putq (blobs module) 63  
 putq (file class module) 322  
 putq (file I/O module) 377  
 putq (socket module) 783

putq (stderr module) 823	putu128Size (socket module) 790
putq (stdout module) 905	putu128Size (stderr module) 859
putQword (blobs module) 58	putu128Size (stdout module) 941
putr32 (blobs module) 64	putu16 (blobs module) 63
putr32 (file class module) 331	putu16 (file class module) 327
putr32 (file I/O module) 420	putu16 (file I/O module) 405
putr32 (socket module) 792	putu16 (socket module) 788
putr32 (stdout module) 947	putu16 (stderr module) 850
putr64 (blobs module) 64	putu16 (stdout module) 933
putr64 (file class module) 331	putu16Size (blobs module) 63
putr64 (file I/O module) 421	putu16size (file class module) 327
putr64 (socket module) 792	putu16size (socket module) 789
putr64 (stderr module) 866	putu16Size (stderr module) 851
putr64 (stdout module) 948	putu16Size (stdout module) 934
putr80 (blobs module) 64	putu32 (blobs module) 63
putr80 (file class module) 331	putu32 (file class module) 327
putr80 (file I/O module) 423	putu32 (file I/O module) 408
putr80 (socket module) 793	putu32 (socket module) 789
putr80 (stderr module) 867	putu32 (stderr module) 853
putr80 (stdout module) 949	putu32 (stdout module) 936
puts (blobs module) 63	putu32Size (blobs module) 63
puts (file class module) 319	putu32size (file class module) 327
puts (file I/O module) 362	putu32size (socket module) 789
puts (socket module) 780	putu32Size (stderr module) 854
puts (stderr module) 810	putu32Size (stdout module) 937
puts (stdout module) 892	putu64 (blobs module) 63
putsSize (blobs module) 63	putu64 (file class module) 328
putssize (file class module) 319	putu64 (file I/O module) 411
putssize (file I/O module) 363	putu64 (socket module) 789
putsSize (socket module) 780	putu64 (stderr module) 855
putsSize (stderr module) 811	putu64 (stdout module) 938
putsSize (stdout module) 893	putu64Size (blobs module) 64
puttb (blobs module) 63	putu64size (file class module) 328
puttb (file class module) 322	putu64size (file I/O module) 412
puttb (file I/O module) 380	putu64size (socket module) 789
puttb (socket module) 784	putu64Size (stderr module) 856
puttb (stderr module) 827	putu64Size (stdout module) 939
puttb (stdout module) 909	putu8 (blobs module) 63
putTbyte (blobs module) 59	putu8 (file class module) 326
putu128 (blobs module) 64	putu8 (file I/O module) 402
putu128 (file class module) 328	putu8 (socket module) 788
putu128 (file I/O module) 413	putu8 (stderr module) 847
putu128 (socket module) 790	putu8 (stdout module) 930
putu128 (stderr module) 858	putu8Size (blobs module) 63
putu128 (stdout module) 941	putu8size (file class module) 327
putu128Size (blobs module) 64	putu8size (file I/O module) 404
putu128size (file class module) 328	putu8size (socket module) 788
putu128Size (file I/O module) 414	putu8Size (stderr module) 849

- putu8Size (stdout module) 932
- putw (blobs module) 63
- putw (file class module) 320
- putw (file I/O module) 369
- putw (socket module) 782
- putw (stderr module) 817
- putw (stdout module) 899
- putWord (blobs module) 58
- putz (socket module) 780
- putzSize (socket module) 781
- Q
  - qSize (conversions module) 94
  - qToBuf (conversions module) 104
  - qToStr (conversions module) 124
- R
  - r32ToBuf (conversions module) 239
  - r32ToStr (conversions module) 241
  - r64ToBuf (conversions module) 238
  - r64ToStr (conversions module) 240
  - r80ToBuf (conversions module) 237
  - r80ToStr (conversions module) 240
  - Radio button sets 512
  - Radio button styles 512
  - Radio buttons 511
  - Radio buttons (HOWL) 511
  - Radio set buttons (HOWL) 513
  - Random numbers 741
  - rangeChar (character sets) 263
  - rbrk (strings module) 1000
  - rbrk2 (strings module) 1000
  - rbrk3 (strings module) 1000
  - rchpos (blobs module) 48
  - rchpos (strings module) 996
  - rchpos2 (blobs module) 48
  - rchpos2 (strings module) 996
  - rchpos3 (blobs module) 48
  - rchpos3 (strings module) 997
  - rcursor (blobs module) 40
  - RDL (Remote procedure call Declaration Language) 746
  - read (blobs module) 60
  - read (file class module) 333
  - read (file I/O module) 425
  - read (socket module) 794
  - read (stdin module) 872
  - readAt (blobs module) 60
  - readLn (blobs module) 64
  - ReadLn (file class module) 333
  - ReadLn (file I/O module) 426
  - readLn (socket module) 795
  - ReadLn (stdin module) 873
  - Real to string output using decimal notation 1091
  - Real to string output using scientific notation 1089
  - realloc (blobs module) 38
  - realloc (memory module) 673
  - realloc1 (memory module) 673
  - realloc2 (memory module) 673
  - Rectangles (HOWL) 514
  - recv (socket module) 763
  - recvfrom (socket module) 763
  - reduce (arrays module) 19
  - reduction (arrays module) 19
  - releaseSemaphore (threads module) 1112
  - Remote Procedure Call Protocol 745
  - Remote procedure call servers 751
  - Remote procedure calls (RPCs) 745
  - Remote procedure types 745
  - removeChar (character sets) 272
  - removeStr (character sets) 275
  - removeStr2 (character sets) 276
  - rename (filesystem module) 464
  - reset (blobs module) 42
  - resize (HOWL) 528
  - restart (timer class) 1135
  - restoreCursor (console module) 76
  - reverse (lists module) 619
  - reverse (strings module) 1037
  - reverse1 (strings module) 1037
  - reverse2 (strings module) 1037
  - reverse32, reverse16, reverse8 (bits module) 26
  - rfindInCset (strings module) 1004
  - rfindInCset2 (strings module) 1004
  - rfindInCset3 (strings module) 1005
  - right (console module) 75
  - rindex (blobs module) 46
  - rindex (strings module) 991
  - rindex2 (blobs module) 46
  - rindex2 (strings module) 991
  - rindex3 (blobs module) 46
  - rindex3 (strings module) 991
  - rindexStr (blobs module) 46
  - rindexStr2 (blobs module) 46
  - rindexStr3 (blobs module) 46

rmdir (filesystem module) 465  
 rmv1stChar1 (strings module) 1014  
 rmv1stChar2 (strings module) 1015  
 rmv1stWord3 (strings module) 1020  
 rmvLastChar1 (strings module) 1015  
 rmvLastChar2 (strings module) 1016  
 rmvLastWord2 (strings module) 1020  
 rmvLastWord3 (strings module) 1021  
 rmvTrailingSpaces1 (strings module) 979  
 rmvTrailingSpaces2 (strings module) 980  
 roman (conversions module) 245  
 Round rectangles (HOWL) 514  
 RPC (remote procedure calls) 745  
 RPC Declaration Language 746  
 RPC Servers 751  
 rskipInCset (strings module) 1001  
 rskipInCset2 (strings module) 1001  
 rskipInCset3 (strings module) 1002  
 rspan (strings module) 999  
 rspan2 (strings module) 999  
 rspan3 (strings module) 999  
 Running (timer class) 1134  
 Run-time traits (stl) 956

## S

save (blobs module) 55  
 saveCursor (console module) 76  
 Scanning functions (blobs module) 45  
 Scroll bars (HOWL) 515, 577  
 scrollDown (console module) 81  
 scrollUp (console module) 80  
 search (lists module) 620  
 \_sec (math module) 650  
 sec (math module) 650  
 sec32 (math module) 650  
 sec64 (math module) 650  
 sec80 (math module) 650  
 secsBetweenTimes (time class module) 1130  
 secsBetweenTimes (time module) 1119  
 select (socket module) 764  
 Semaphores 1110  
 send (socket module) 765  
 sendto (socket module) 765  
 Server applications (socket module) 771  
 server\_t.close 775  
 server\_t.create 775  
 server\_t.destroy 775  
 setAdrs (socket module) 777  
 setAttrs (console module) 81  
 setDelimiters (conversions module) 89  
 setEvent (threads module) 1107  
 set\_exStyle (HOWL) 528  
 set\_focus (HOWL) 529  
 setFormat (date/time module) 290  
 setFormat (time module) 1123  
 set\_height (HOWL) 528  
 setLength (blobs module) 39  
 setMaxLen (blobs module) 40  
 set\_onHeap (HOWL) 526  
 set\_parentHandle (HOWL) 526, 528  
 setPort (socket class module) 778  
 setrCursor (blobs module) 41  
 setSeparator (date/time module) 290  
 set\_style (HOWL) 528  
 setTimeout (socket class module) 777  
 setTimeout (socket module) 768  
 setTimeout2 (socket class module) 777  
 setTLS (threads module) 1106  
 setUnderscores (conversions module) 85  
 setunion (character sets) 268  
 setwCursor (blobs module) 41  
 setWhiteSpace (patterns module) 727  
 set\_width (HOWL) 528  
 setWordDelims (patterns module) 726  
 set\_x (HOWL) 528  
 set\_y (HOWL) 528  
 Shell commands 681  
 shl1 (math module) 639  
 shlq (math routine) 630  
 show (HOWL) 526, 529  
 shrl (math module) 640  
 shrq (math module) 631  
 Signed integer concatenation functions 1068  
 Signed integer input (socket module) 796  
 Signed integer numeric output (socket module) 785  
 \_sin (math module) 641  
 sin (math module) 641  
 sin32 (math module) 641  
 sin64 (math module) 641  
 sin80 (math module) 641  
 \_sincos (math module) 645  
 sincos (math module) 645  
 sincos32 (math module) 645  
 sincos64 (math module) 645

sincos80 (math module) 645  
 SingleStep (exceptions module) 309  
 size (filesystem module) 462  
 size (memory module) 675  
 skip (patterns module) 708  
 skipInCset (strings module) 1000  
 skipInCset2 (strings module) 1000  
 skipInCset3 (strings module) 1001  
 sleep (os module) 682  
 sock.a\_adrsToStr 761  
 sock.accept 762  
 sock.adrsToStr 761  
 sock.bind 762  
 sock.close 763  
 sock.connect 763  
 sock.fd\_clr 769  
 sock.fd\_iset 769  
 sock.fd\_set 769  
 sock.fd\_zero 769  
 sock.gethostbyaddr 769  
 sock.gethostbyname 768  
 sock.gethostname 768  
 sock.getpeername 769  
 sock.getsockname 769  
 sock.listen 763  
 sock.recv 763  
 sock.recvfrom 763  
 sock.select 764  
 sock.send 765  
 sock.sendto 765  
 sock.setTimeout 768  
 sock.socket 765  
 sock.socketCleanup 761  
 sock.socketInit 761  
 sock.strToAdrs 762  
 socket (socket module) 765  
 Socket class operations 775  
 Socket classes 770  
 socketCleanup (socket module) 761  
 socketInit (socket class module) 761  
 Sockets 761  
 sort (lists module) 620  
 span (strings module) 999  
 span2 (strings module) 999  
 span3 (strings module) 999  
 spread2 (strings module) 1024– 1025  
 spread3 (strings module) 1025  
 StackOverflow (exceptions module) 309  
 Standard Error output routines 803  
 Standard input routines 871  
 Standard output routines 885  
 Standard Template Library 953  
 start (timer class) 1135  
 stat (memory module) 675  
 stderr.hhf 803  
 stderr.newln 805  
 stderr.put 869  
 stderr.putb 813  
 stderr.putbool 805  
 stderr.putc 806  
 stderr.putcset 809  
 stderr.putcSize 808  
 stderr.putd 820  
 stderr.pute32 861  
 stderr.pute64 862  
 stderr.pute80 863  
 stderr.puth128 831  
 stderr.puth128Size 832  
 stderr.puth16 818  
 stderr.puth16Size 819  
 stderr.puth32 821  
 stderr.puth32Size 822  
 stderr.puth64 824  
 stderr.puth64Size 825  
 stderr.puth80 827  
 stderr.puth80Size 828  
 stderr.puth8Size 815  
 stderr.puti128 845  
 stderr.puti128Size 845  
 stderr.puti16 837  
 stderr.puti16Size 838  
 stderr.puti32 840  
 stderr.puti32Size 841  
 stderr.puti64 842  
 stderr.puti64Size 843  
 stderr.puti8 834  
 stderr.puti8Size 836  
 stderr.putl 830  
 stderr.putq 823  
 stderr.putr64 866  
 stderr.putr80 867  
 stderr.puts 810  
 stderr.putsSize 811  
 stderr.puttb 827  
 stderr.putu128 858  
 stderr.putu128Size 859

stderr.putu16	850	stdout.pute32	944
stderr.putu16Size	851	stdout.pute64	944
stderr.putu32	853	stdout.pute80	945
stderr.putu32Size	854	stdout.puth128	914
stderr.putu64	855	stdout.puth128Size	915
stderr.putu64Size	856	stdout.puth16	900
stderr.putu8	847	stdout.puth16Size	901
stderr.putu8Size	849	stdout.puth32	903
stderr.putw	817	stdout.puth32Size	904
stderr.write	804	stdout.puth64	906
stdin.a_gets	875	stdout.puth64Size	907
stdin.eoln	873	stdout.puth8	896
stdin.eoln2	873	stdout.puth80	910
stdin.FlushInput	871	stdout.puth80Size	911
stdin.get	883	stdout.puth8Size	897
stdin.getc	874	stdout.puti128	928
stdin.getf	883	stdout.puti128Size	928
stdin.geth128	877	stdout.puti16	920
stdin.geth16	876	stdout.puti16Size	921
stdin.geth32	876	stdout.puti32	923
stdin.geth64	877	stdout.puti32Size	924
stdin.geth8	875	stdout.puti64	925
stdin.geti128	880	stdout.puti64Size	926
stdin.geti16	878	stdout.puti8	917
stdin.geti32	879	stdout.puti8Size	919
stdin.geti64	879	stdout.putl	913
stdin.geti8	878	stdout.putq	905
stdin.gets	874	stdout.putr32	947
stdin.getu128	882	stdout.putr64	948
stdin.getu16	881	stdout.putr80	949
stdin.getu32	881	stdout.puts	892
stdin.getu64	882	stdout.putsSize	893
stdin.getu8	880	stdout.puttb	909
stdin.handle	871	stdout.putu128	941
stdin.hhf	871	stdout.putu128Size	941
stdin.peekc	874	stdout.putu16	933
stdin.read	872– 873	stdout.putu16Size	934
stdin.ReadLn	873	stdout.putu32	936
stdout.handle	803, 885	stdout.putu32Size	937
stdout.hhf	885	stdout.putu64	938
stdout.newln	887	stdout.putu64Size	939
stdout.put	951	stdout.putu8	930
stdout.putb	895	stdout.putu8Size	932
stdout.putbool	887	stdout.putw	899
stdout.putc	888	stdout.write	886
stdout.putcset	891	stl.elementsAreObjects_c	956– 957, 959
stdout.putcSize	890	stl.fastAppend_c	956– 957, 959
stdout.putd	902	stl.fastElementSwap_c	956– 957, 960



stl.fastInsert\_c 956– 957, 959  
 stl.fastPrepend\_c 956– 957, 959  
 stl.fastRemove\_c 956– 957, 959  
 stl.fastSearch\_c 956– 957, 960  
 stl.fastSwap\_c 956– 957, 960  
 stl.IsArray\_c 956– 957  
 stl.isArray\_c 958  
 stl.isContainer\_c 956– 957  
 stl.isDeque\_c 956– 958  
 stl.isList\_c 956– 958  
 stl.isRandomAccess\_c 956– 958  
 stl.isTable\_c 956– 958  
 stl.isVector\_c 956– 957  
 stl.supportsAppend\_c 957– 958  
 stl.supportsCompare\_c 956– 958  
 stl.supportsCursor\_c 956– 957, 959  
 stl.supportsElementSwap\_c 956– 957, 959  
 stl.supportsForEach\_c 956– 957, 959  
 stl.supportsInsert\_c 956– 958  
 stl.supportsObjSwap\_c 956– 957, 959  
 stl.supportsOutput\_c 956– 958  
 stl.supportsPrepend\_c 956– 958  
 stl.supportsRemove\_c 956– 958  
 stl.supportsrForEach\_c 956– 957  
 stl.supportsSearch\_c 956– 957, 959  
 stl.supportsSwap\_c 956– 957  
 stl.vector 953  
 stop (timer class) 1135  
 str.a\_cat 1040  
 str.a\_catbuf 1045  
 str.a\_catbuf2 1045  
 str.a\_catbuf3 1045  
 str.a\_catsub 1043  
 str.a\_catz 1042  
 str.a\_columnize2 1022  
 str.a\_columnize3 1022  
 str.a\_cpy 965  
 str.a\_cpyz 965  
 str.a\_delete 973  
 str.a\_delLeadingSpaces 975  
 str.a\_delTrailingSpaces 976  
 str.a\_deTab2 1026  
 str.a\_deTab3 1027  
 str.a\_enTab2 1030  
 str.a\_enTab3 1030  
 str.a\_first 967  
 str.a\_getField2 1011  
 str.a\_getField3 1012  
 str.a\_insert 972  
 str.a\_last 969  
 str.alloc (memory/string module) 679  
 str.a\_lower 1035  
 str.a\_reverse 1036  
 str.a\_rmv1stWord1 1016  
 str.a\_rmv1stWord2 1017  
 str.a\_rmvLastWord1 1018  
 str.a\_rmvLastWord2 1018  
 str.a\_rmvTrailingSpaces 979  
 str.a\_spread2 1024  
 str.a\_substr 966  
 str.a\_translate 1038  
 str.a\_trim 978  
 str.a\_truncate 970  
 str.a\_upper 1034  
 str.brk 999  
 str.brk2 999  
 str.brk3 1000  
 str.cat 1041  
 str.cat2 1041  
 str.cat3 1041  
 str.catb 1053  
 str.catbool 1047  
 str.catbuf 1046  
 str.catbuf2 1046  
 str.catbuf3a 1046  
 str.catbuf3b 1046  
 str.catbuf4 1046  
 str.cate 1048  
 str.catecset 1050  
 str.cateSize 1049  
 str.catd 1059  
 str.cate32 1089  
 str.cate64 1090  
 str.cate80 1090  
 str.cath128 1066  
 str.cath128Size 1067  
 str.cath16 1057  
 str.cath16Size 1058  
 str.cath32 1060  
 str.cath32Size 1060  
 str.cath64 1062  
 str.cath64Size 1062  
 str.cath80 1064  
 str.cath80Size 1064  
 str.cat128 1076

str.catil28Size	1077	str.deTab2	1027
str.catil6	1071	str.deTab3a	1028
str.catil6Size	1072	str.deTab3b	1028
str.cati32	1073	str.deTab4	1029
str.cati32Size	1073	str.enTab2	1031
str.cati64	1075	str.enTab3a	1032
str.cati64Size	1075	str.enTab3b	1032
str.cati8	1068	str.enTab4	1033
str.cati8Size	1070	str.eq	980
str.catl	1066	str.findInCset	1003
str.catq	1061	str.findInCset2	1003
str.catr32	1091	str.findInCset3	1003
str.catr64	1092	str.first	968
str.catr80	1093	str.first2	968
str.cats	1051	str.first3	968
str.catsSize	1052	str.free (memory/string module)	679
str.catsub	1043	str.ge	983
str.catsub4	1044	str.getField3	1013
str.catsub5	1044	str.getField4	1013
str.cattb	1064	str.getref	679
str.catu128	1087	str.gt	982
str.catu128Size	1087	str.ichpos	995
str.catu16	1081	str.ichpos2	995
str.catu16Size	1082	str.ichpos3	995
str.catu32	1083	str.ieq	983
str.catu32Size	1083	str.ige	985
str.catu64	1085	str.igt	985
str.catu64Size	1085	str.iindex	989
str.catu8	1079	str.iindex3	990
str.catu8Size	1080	str.ile	985
str.catw	1056	str.ilt	984
str.catz	1042	str.index	988
str.charInStr	1010	str.index2	988
str.chpos	994	str.index3	988
str.chpos2	994	str.ine	984
str.chpos3	994	str.init	963
str.columnize3	1023	str.insert	972
str.columnize4	1023	str.insert3	972
str.cpy	965	str.insert4	973
str.cpyz	966	str.irchpos	997
str.delete	974	str.irchpos2	998
str.delete3	974	str.irchpos3	998
str.delete4	974	str.irindex	992
str.delLeadingSpaces	975– 976	str.irindex2	992
str.delLeadingSpaces1	976	str.irindex3	993
str.delTrailingSpaces	977	str.isInHeap (memory/string module)	679
str.delTrailingSpaces1	977	str.last	969
str.delTrailingSpaces2	977	str.last2	969

str.last3 970  
 str.le 982  
 str.length 964  
 str.lower 1036  
 str.lower1 1036  
 str.lower2 1036  
 str.lt 981  
 str.ne 981  
 str.newref (memory/string module) 679  
 str.prefix 986  
 str.prefix3 987  
 str.put 1095  
 str.rbrk 1000  
 str.rbrk2 1000  
 str.rbrk3 1000  
 str.rchpos 996  
 str.rchpos2 996  
 str.rchpos3 997  
 str.realloc (memory/string module) 679  
 str.reverse 1037  
 str.reverse1 1037  
 str.reverse2 1037  
 str.rfindInCset 1004  
 str.rfindInCset2 1004  
 str.rfindInCset3 1005  
 str.rindex 991  
 str.rindex2 991  
 str.rindex3 991  
 str.rmv1stChar1 1014  
 str.rmv1stChar2 1015  
 str.rmv1stWord2rmv1stWord2 (strings module) 1019  
 str.rmv1stWord3 1020  
 str.rmvLastChar1 1015  
 str.rmvLastChar2 1016  
 str.rmvLastWord2 1020  
 str.rmvLastWord3 1021  
 str.rmvTrailingSpaces1 979  
 str.rmvTrailingSpaces2 980  
 str.rskipInCset 1001  
 str.rskipInCset2 1001  
 str.rskipInCset3 1002  
 str.rspan 999  
 str.rspan2 999  
 str.rspan3 999  
 str.skipInCset 1000  
 str.skipInCset2 1000  
 str.skipInCset3 1001  
 str.span 999  
 str.span2 999  
 str.span3 999  
 str.spread2 1025  
 str.spread3 1025  
 str.strvar 963  
 str.substr 967  
 str.talloc (memory/string module) 679  
 str.tokenCnt1 1005  
 str.tokenCnt2 1006  
 str.tokenInStr 1009  
 str.tokenInStr2 1009  
 str.tokenize 1006  
 str.tokenize3 1006  
 str.tokenize4 1008  
 str.translate 1039  
 str.translate3 1039  
 str.translate4 1039  
 str.trim 978  
 str.trim1 978  
 str.trim2 978  
 str.truncate 971  
 str.truncate2 971  
 str.truncate3 971  
 str.upper 1034  
 str.upper1 1034  
 str.upper2 1035  
 str.wordInStr 1010  
 String allocation macros and functions 963  
 String assignment 964  
 String comparison functions 980  
 String concatenation functions 1040  
 String concatenation functions 1048  
 String conversion functions 1033  
 String deletion 972  
 String extraction functions (pattern matching module) 726  
 String formatting routines 1021  
 String insertion and deletion 972  
 String length calculations 964  
 String matching functions 724  
 String memory allocation 678  
 String parsing functions 1005  
 String Searching Functions 986  
 String value concatenation functions 1047  
 StringIndexError (exceptions module) 305  
 StringOverflow (exceptions module) 305  
 Strings 961

strings.hhf 961  
 strToAdrs (socket module) 762  
 strToFlt (conversions module) 245  
 strToh128 (conversions module) 155  
 strToh16 (conversions module) 152  
 strToh32 (conversions module) 153  
 strToh64 (conversions module) 154  
 strToh8 (conversions module) 151  
 strToi128 (conversions module) 191  
 strToi16 (conversions module) 188  
 strToi32 (conversions module) 189  
 strToi64 (conversions module) 190  
 strToi8 (conversions module) 187  
 strTou128 (conversions module) 228  
 strTou16 (conversions module) 224  
 strTou32 (conversions module) 225  
 strTou64 (conversions module) 226  
 strTou8 (conversions module) 223  
 subBlob (blobs module) 44  
 subDays (date module) 288  
 subDays (date/time module) 299  
 subHours (time class module) 1130  
 subHours (time module) 1120  
 subl (math module) 632  
 subMins (time class module) 1130  
 subMins (time module) 1120  
 subMonths (date/time module) 288, 299  
 subq (math routine) 624  
 subSecs (time class module) 1130  
 subSecs (time module) 1120  
 subset (character sets) 253  
 substr (strings module) 967  
 Substring functions 966  
 subYears (date/time module) 288, 299  
 superset (character sets) 254  
 supportsAppend\_c (stl) 957– 958  
 supportsCompare\_c (stl) 956– 958  
 supportsCursor\_c (stl) 956– 957, 959  
 supportsElementSwap\_c (stl) 956– 957, 959  
 supportsForEach\_c (stl) 956– 957, 959  
 supportsInsert\_c (stl) 956– 958  
 supportsObjSwap\_c (stl) 956– 957, 959  
 supportsOutput\_c (stl) 956– 958  
 supportsPrepend\_c (stl) 956– 958  
 supportsRemove\_c (stl) 956– 958  
 supportsrForEach\_c (stl) 956– 957  
 supportsrForeach\_c (stl) 959

supportsSearch\_c (stl) 956– 957, 959  
 supportsSwap\_c (stl) 957  
 switch macro 1097  
 Synchronous remote procedures 745  
 system (os module) 681  
 System icons (HOWL) 506  
 System time 1122

## T

Tab pages (HOWL) 597  
 Tabbed forms 545  
 Table (stl) 960  
 Table class 1099  
 tableNode\_t 1099  
 Tables 1099  
 tables.hhf 1099  
 table\_t.create 1099– 1100  
 table\_t.destroy 1099– 1100  
 table\_t.getNode 1099, 1102  
 table\_t.item 1099  
 table\_t.lookup 1099, 1101  
 talloc (memory module) 674  
 \_tan (math module) 644  
 tan (math module) 644  
 tan32 (math module) 644  
 tan64 (math module) 644  
 tan80 (math module) 644  
 tbSize (conversions module) 94  
 tbToBuf (conversions module) 105  
 tbToStr (conversions module) 126  
 Templates (stl) 954  
 TenToX (math module) 659  
 \_tenToX (math module) 659  
 tenToX32 (math module) 659  
 tenToX64 (math module) 659  
 tenToX80 (math module) 659  
 Text (HOWL) 592  
 Text editor widgets (HOWL) 516  
 Thread local storage 1105  
 thread.create 1103  
 thread.createCriticalSection 1109  
 thread.createEvent 1107  
 thread.createSemaphore 1110  
 thread.createTLS 1105  
 thread.deleteCriticalSection 1109  
 thread.deleteEvent 1107  
 thread.deleteSemaphore 1111  
 thread.enterCriticalSection 1109  
 thread.getCurrentThreadHandle 1105

thread.getTLS 1106  
 thread.leaveCriticalSection 1110  
 thread.releaseSemaphore 1112  
 thread.setEvent 1107  
 thread.setTLS 1106  
 thread.waitForEvent 1108  
 thread.waitSemaphore 1111  
 Threads 1103  
 threads.hhf 1103  
 Three-state check boxes 500  
 Time arithmetic 1119  
 Time conversions 1116  
 Time delay functions 681  
 Time functions 1113  
 Time string conversions 1123  
 time.addHours 1121  
 time.addMins 1121  
 time.addSecs 1121  
 time.a\_toString 1123  
 time.curTime 1122, 1130  
 time.durationToSecs 1116  
 time.fromSecs 1117  
 time.fromUnixTime 1118  
 time.fromWinFileTime 1119  
 time.\_isValid 1115  
 time.isValid 1115  
 time.pack 1116  
 time.secsBetweenTimes 1119  
 time.setFormat 1123  
 time.subHours 1120  
 time.subMins 1120  
 time.subSecs 1120  
 time.timerec 1113  
 time.\_toSecs 1117  
 time.toSecs 1117  
 time.toString 1123  
 time.toUnixTime 1117  
 time.toWinFileTime 1118  
 time.unpack 1116  
 time.utcTime 1122, 1130  
 time.validate 1114  
 TimeOverflow (exceptions module) 309  
 Timer functions 1133  
 Timer procedures and methods 1134  
 Timer widgets (HOWL) 519  
 timer.hhf 1133  
 timerec (date/time module) 1113  
 Timers (HOWL) 601  
 timer\_t 1133  
 timer\_t.Accumulated 1134  
 timer\_t.checkPoint 1135  
 timer\_t.create 1134  
 timer\_t.DateStarted 1134  
 timer\_t.DateStopped 1134  
 timer\_t.msStarted 1134  
 timer\_t.msStopped 1134  
 timer\_t.restart 1135  
 timer\_t.Running 1134  
 timer\_t.start 1135  
 timer\_t.stop 1135  
 timer\_t.TimeStarted 1134  
 timer\_t.TimeStopped 1134  
 TimeStarted (timer class) 1134  
 TimeStopped (timer class) 1134  
 today (date/time module) 289, 299  
 toJulian (date/time module) 283, 298  
 tokenCnt1 (strings module) 1005  
 tokenCnt2 (strings module) 1006  
 tokenInStr (strings module) 1009  
 tokenInStr2 (strings module) 1009  
 tokenize (strings module) 1006  
 tokenize3 (strings module) 1006  
 tokenize4 (strings module) 1008  
 toLower (chars module) 65  
 toNativePath1 (filesystem module) 460  
 toNativePath2 (filesystem module) 460  
 TooManyCmdLnParms (exceptions module) 306  
 toSecs (time class module) 1129  
 \_toSecs (time module) 1117  
 toSecs (time module) 1117  
 toString (date/time module) 291, 299  
 toString (time class module) 1131  
 toString (time module) 1123  
 toUnixPath (filesystem module) 457  
 toUnixPath2 (filesystem module) 458  
 toUnixTime (time module) 1117  
 toUpper (chars module) 65  
 toWin32Path1 (filesystem module) 459  
 toWin32Path2 (filesystem module) 459  
 toWinFileTime (time module) 1118  
 Track bar widgets (HOWL) 516  
 Track bars (HOWL) 577  
 Trait constants 957  
 traits (stl classes) 955  
 Transcendental arithmetic operations 640

translate (strings module) 1039  
 translate3 (strings module) 1039  
 translate4 (strings module) 1039  
 transpose (arrays module) 20  
 trim (strings module) 978  
 trim1 (strings module) 978  
 trim2 (strings module) 978  
 truncate (strings module) 971  
 truncate2 (strings module) 971  
 truncate3 (strings module) 971  
 \_twoToX (math module) 658  
 twoToX (math module) 658  
 twoToX32 (math module) 658  
 twoToX64 (math module) 658  
 twoToX80 (math module) 658  
 typeName field in stl classes 960

## U

u128Size (conversions module) 196  
 u128ToBuf (conversions module) 202  
 u128ToStr (conversions module) 216  
 \_u16Size (conversions module) 193  
 u16Size (conversions module) 194  
 u16ToBuf (conversions module) 199  
 u16ToStr (conversions module) 208  
 \_u32Size (conversions module) 193  
 u32Size (conversions module) 195  
 u32ToBuf (conversions module) 200  
 u32ToStr (conversions module) 211  
 u64Size (conversions module) 195  
 u64ToBuf (conversions module) 201  
 u64ToStr (conversions module) 214  
 \_u8Size (conversions module) 193  
 u8Size (conversions module) 193  
 u8ToBuf (conversions module) 197  
 u8ToStr (conversions module) 204  
 Underscore control 84  
 unionChar (character sets) 271  
 unionStr (character sets) 273  
 unionStr2 (character sets) 274  
 UnknownException (exception) 305, 308  
 unpack (date/time module) 283  
 unpack (time module) 1116  
 Unsigned integer concatenation functions 1078  
 Unsigned integer conversions 192  
 Unsigned integer numeric output (socket module) 788  
 up (console module) 74

Up/Down arrow widgets (HOWL) 517–518  
 Up/Down arrows (HOWL) 587  
 upper (strings module) 1034  
 upper1 (strings module) 1034  
 upper2 (strings module) 1035  
 upToChar (patterns module) 716  
 upToCset (patterns module) 711  
 upToiChar (patterns module) 720  
 upToiStr (patterns module) 725  
 upToStr (patterns module) 725  
 utc (date/time module) 289  
 utcTime (time class module) 1130  
 utcTime (time module) 1122

## V

v (args module) 10  
 validate (date/time module) 282, 298  
 validate (time class module) 1129  
 validate (time module) 1114  
 ValueOutOfRange (exceptions module) 305  
 Variable length hexadecimal numeric to buffer conversions 108  
 Vector (stl) 960  
 Views (HOWL) 597  
 visible (HOWL) 525

## W

wabsEditBox\_t 535  
 waitForEvent (threads module) 1108  
 waitSemaphore (threads module) 1111  
 wBase\_t 521, 523  
 wBitmap 510  
 wBitmap\_t 548  
 wButton\_t 530  
 wCheckable\_t 532  
 wCheckBox 501  
 wCheckBox3 502  
 wCheckBox3LT 502  
 wCheckBox3LT\_t 559  
 wCheckBox3\_t 558  
 wCheckBoxLT 502  
 wCheckBoxLT\_t 560  
 wCheckBox\_t 557  
 wClickable\_t 529  
 wComboBox 502  
 wComboBox\_t 574  
 wContainer\_t 539  
 wcursor (blobs module) 41

- wDragListBox 503
- wDragListBox\_t 573
- wEditBox 504
- wEditBox\_t 566
- wEllipse 505
- wEllipse\_t 550
- wFilledFrame\_t 534
- wFont\_t 592
- wForm 486, 495– 496
- wForm\_t 541
- wGroupBox 506
- wGroupBox\_t 547
- wIcon 506
- wIcon\_t 591
- widgetProc 493
- WidthTooBig (exceptions module) 306
- Window objects (HOWL) 599
- window\_t 599
- wLabel 507
- wLabel\_t 594
- wListBox 508
- wListBox\_t 570
- wMainMenu 497
- wMenuItem 498
- wMenuItem\_t 543
- wMenuSeparator 498
- wMenu\_t 543
- wordInStr (strings module) 1010
- wPasswdBox 508
- wPasswdBox\_t 567
- wPie 509
- wPie\_t 551
- wPolygon 509
- wPolygon\_t 553
- wProgressBar 511
- wProgressBar\_t 576
- wPushButton 493, 495, 511
- wPushButton\_t 561
- wRadioButton 511
- wRadioButtonLT 512
- wRadioButtonLT\_t 562
- wRadioButton\_t 561
- wRadioSet 512
- wRadioSetButton 513
- wRadioSetButtonLT 513
- wRadioSetButtonLT\_t 565
- wRadioSetButton\_t 564
- wRadioSet\_t 563
- wRectangle 514
- wRectangle\_t 555
- write (blobs module) 55
- write (file class module) 317
- write (file I/O module) 354
- write (socket module) 778
- write (stderr module) 804
- write (stdout module) 886
- writeAt (blobs module) 57
- Writing your own pattern matching routines 729
- wRoundRect 514
- wRoundRect\_t 556
- wScrollBar 515
- wScrollBar\_t 578
- wSize (conversions module) 93
- WSorEOS (patterns module) 728
- WSthenEOS (patterns module) 728
- wSubMenu 498
- wSurface\_t 533
- wTab 499
- wTabPage\_t 597
- wTabs\_t 545
- wTextEdit 516
- wTextEdit\_t 568
- wTimer 519
- wTimer\_t 601
- wToBuf (conversions module) 102
- wToStr (conversions module) 119
- wTrackBar 516
- wTrackBar\_t 583
- wType (HOWL) 521, 524
- wUpDown 517– 518
- wUpDown\_t 587, 589
- wView\_t 599
- wVisual\_t 526
- X
- xchgNodes (lists module) 620
- xorl (math module) 638
- xorq (math routine) 629
- Y
- \_YtoX (math module) 661
- ytoX (math module) 661
- YtoX32 (math module) 661
- YtoX64 (math module) 661
- YtoX80 (math module) 661
- Z
- zalloc (memory module) 672

zcmp (zstrings module)	1139	zeroOrOneiChar (patterns module)	720
zeroOrMoreChar (patterns module)	716	zeroOrOnePat (patterns module)	710
zeroOrMoreCset (patterns module)	712	Zero-terminated strings	1137
zeroOrMoreiChar (patterns module)	721	zstr.cat	1140
zeroOrMorePat (patterns module)	710	zstr.cpy	1140
zeroOrMoreWS (patterns module)	727	zstr.len	1138
zeroOrOneChar (patterns module)	716	zstr.zcmp	1139
zeroOrOneCset (patterns module)	711	zstring.hhf	1137