31 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).

31.1 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;
```

31.2 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 maxium number of charactera 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;
```

31.3 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 );
```

31.4 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, newHLAStr );



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;
```

31.5 Substring Functions

The HLA Standard Library provides four families of functions that extract a portion of some sring (that is, a substring): the *substr, first, last,* and *truncate* families of functions. These functions dffer 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 specifed (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 sufficent 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 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 sufficent 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 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 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 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 sufficent 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 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 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 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 sufficent 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 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 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;
```

31.6 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 sufficent 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 creates 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 *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 sufficent 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 creates 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 creates 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 sufficent 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 sufficent 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 sufficent 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 sufficent 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.delTrailingSpace1s* 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;
```

31.7 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:
```

31.8 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 Node: 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* atarting 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-insenstive 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-insenstive search for *subst* within *baseStr*. If there are multiple occurrences of *subst* within *baseStr*, this function locates the last one in *baseStr* finds 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-insenstive 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-insenstive search for *src* within *baseStr*. If there are multiple occurrences of *src* within *baseStr*, this function locates the last one in *baseStr* finds 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:
```

31.9 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.span 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:
```

31.10 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_rmv1stWord1;
  jnc noWord;

         // Do something with the string pointed at by EAX.

  noWord:
```

procedure str.a_rmv1stWord2(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_rmv1stWord2( 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_rmv1stWord2;
  jnc noWord;

        mov( eax, wordStr );

        // Do something with the strings in EAX and remainder.
            .
            .
            .
        str.free( wordStr );

  noWord:
```

procedure str.a_rmvLastWord1(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_rmvLastWord1( 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_rmvLastWord1;
  jnc noWord;

         mov( eax, wordStr );

         // Do something with the string in EAX.
             .
             .
             .
         str.free( wordStr );

  noWord:
```

procedure str.a_rmvLastWord2(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_rmvLastWord2( 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:
```

31.11 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;


   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
```

31.12 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
```

31.13 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_catsub(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 catsub(string, dword, dword, string);
#macro catsub(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
```

31.14 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.

31.14.1 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;
```

31.14.2 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 justfied 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 equilvalent 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;
```

31.14.3 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 signficant 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);

e *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.catl(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.catl( 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.catl;
```

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;
```

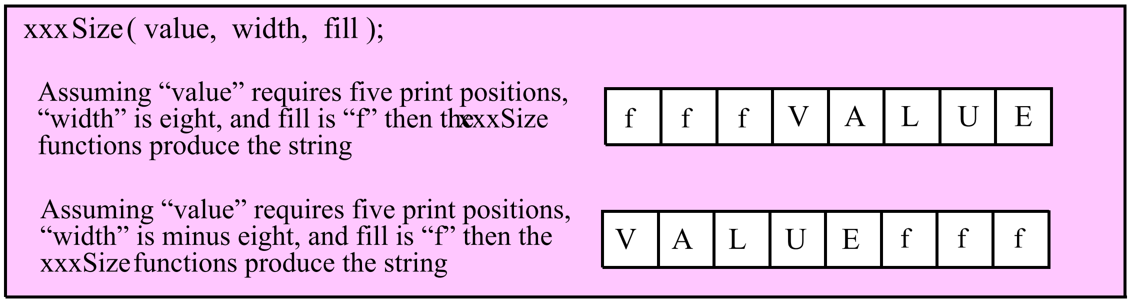
31.14.4 Signed Integer Concatenation Routines

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.cati8Size;
```

procedure str.cati16(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.cati16( dest, wordVar );

// If the word is in a register (AX):

str.cati16( 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.cati16;

// 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.cati16;

// If no register is available, do something
// like the following code:

push( dest );
push( eax );
movzx( wordVar, eax );
push( eax );
call str.cati16;
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.cati16;
```

procedure str.cati16Size(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.cati16Size( 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.cati16Size;

// 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.cati16Size;

// 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.cati16Size;
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.cati16Size;
```

procedure str.cati32(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.cati32( dest, dwordVar );

// If the dword is in a register (EAX):

str.cati32( dest, eax );


HLA low-level calling sequence examples:

push( dest );
push( dwordVar );
call str.cati32;

push( dest );
push( eax );
call str.cati32;
```

procedure str.cati32Size(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.cati128;
```

procedure str.cati128Size(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.

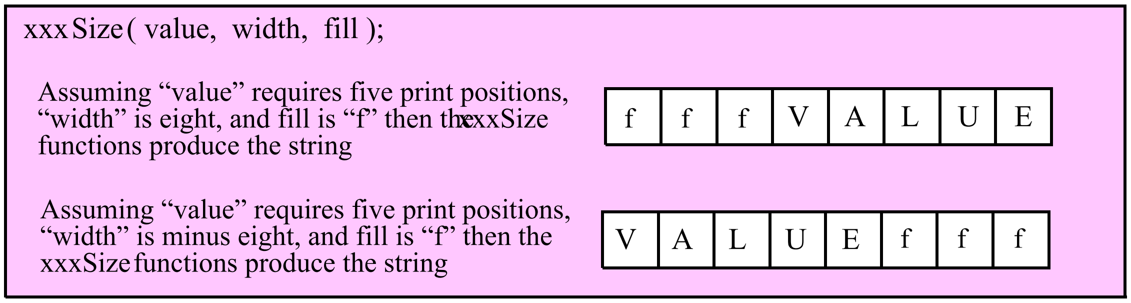
```
HLA high-level calling sequence examples:


str.cati128Size( 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.cati128Size;


// 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.cati128Size;

// 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.cati128Size;

// 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.cati128Size;


// 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.cati128Size;

// 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.cati128Size;
```

31.14.5 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

Assuming “value” requires five print positions, “width” is minus eight, and fill is “f” then the xxxSize functions produce the string

`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;
```

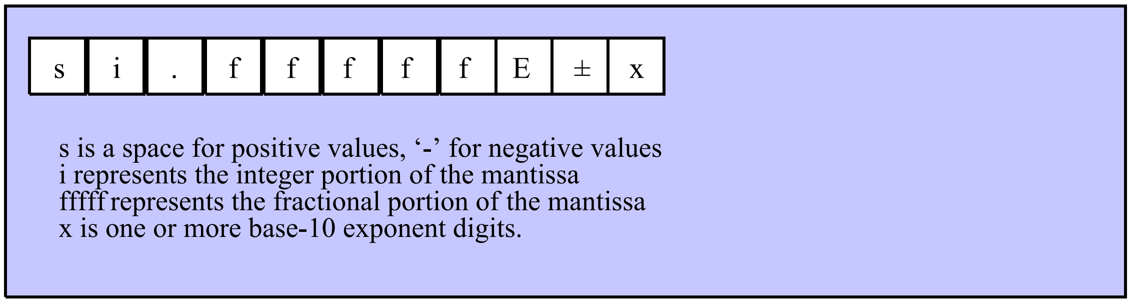
31.15 Floating-Point Concatenation Routines

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.

31.15.1 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 is a space for positive values, '-' for negative values
i represents the integer portion of the mantissa
fffff 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.

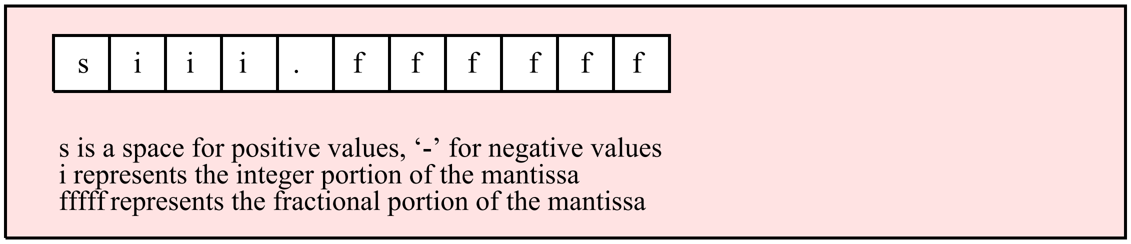
```
HLA high-level calling sequence examples:


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;
```

31.15.2 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 is a space for positive values, '-' for negative values
i represents the integer portion of the mantissa
fffff 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;
```

31.16 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 prefectly 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.