# 1 Passing Parameters to Standard Library Routines

## 1.1 Parameter Passing

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

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

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

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

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

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

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

will emit the following code:

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

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

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

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

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

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

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

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

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

```
call bToBuf;
```

## 1.2 Passing Parameters by Reference and by Value

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

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

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

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

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

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

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

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

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

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

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

someFunction( actualByteParameter );

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

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

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

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

```
someFunction( edi );
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
someStrFunction( eax );
```

## 1.3 Passing Byte Parameters on the Stack

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
someLibraryRoutine( byteVar );
```

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

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

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

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

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

## 1.4 Passing Word Parameters on the Stack

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

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

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

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

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

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

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

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

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

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

```
someLibraryRoutine( wordVar );
```

## 1.5 Passing DWord Parameters on the Stack

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 1.6 Passing QWord Parameters on the Stack

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

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

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

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

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

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

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

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

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

## 1.7 Passing TByte Parameters on the Stack

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

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

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

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

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

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

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

## 1.8 Passing LWord Parameters on the Stack

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

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

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

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

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

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

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