Advanced Parameter Implementation Chapter Four

4.1 Chapter Overview

This chapter discusses advanced parameter passing techniques in assembly language. Both low-level and high-level syntax appears in this chapter. This chapter discusses the more advanced pass by value/result, pass by result, pass by name, and pass by lazy evaluation parameter passing mechanisms. This chapter also discusses how to pass parameters in a low-level manner and describes where you can pass such parameters.

4.2 Parameters

Although there is a large class of procedures that are totally self-contained, most procedures require some input data and return some data to the caller. Parameters are values that you pass to and from a procedure. There are many facets to parameters. Questions concerning parameters include:

- where is the data coming from?
- how do you pass and return data?
- what is the amount of data to pass?

Previous chapters have touched on some of these concepts (see the chapters on beginning and intermediate procedures as well as the chapter on Mixed Language Programming). This chapter will consider parameters in greater detail and describe their low-level implementation.

4.3 Where You Can Pass Parameters

Up to this point we've mainly used the 80x86 hardware stack to pass parameters. In a few examples we've used machine registers to pass parameters to a procedure. In this section we explore several different places where we can pass parameters. Common places are

- in registers,
- in FPU or MMX registers,
- in global memory locations,
- on the stack,
- in the code stream, or
- in a parameter block referenced via a pointer.

Finally, the amount of data has a direct bearing on where and how to pass it. For example, it's generally a bad idea to pass large arrays or other large data structures by value because the procedure has to copy that data onto the stack when calling the procedure (when passing parameters on the stack). This can be rather slow. Worse, you cannot pass large parameters in certain locations; for example, it is not possible to pass a 16-element int32 array in a register.

Some might argue that the only locations you need for parameters are the register and the stack. Since these are the locations that high level languages use, surely they should be sufficient for assembly language programmers. However, one advantage to assembly language programming is that you're not as constrained as a high level language; this is one of the major reasons why assembly language programs can be more efficient than compiled high level language code. Therefore, it's a good idea to explore different places where we can pass parameters in assembly language.

This section discusses six different locations where you can pass parameters. While this is a fair number of different places, undoubtedly there are many other places where one can pass parameters. So don't let this section prejudice you into thinking that this is the only way to pass parameters.

4.3.1 Passing Parameters in (Integer) Registers

Where you pass parameters depends, to a great extent, on the size and number of those parameters. If you are passing a small number of bytes to a procedure, then the registers are an excellent place to pass parameters. The registers are an ideal place to pass value parameters to a procedure. If you are passing a single parameter to a procedure you should use the following registers for the accompanying data types:

Data Size	Pass in this Register
Byte:	al
Word:	ax
Double Word:	eax
Quad Word:	edx:eax

This is, by no means, a hard and fast rule. If you find it more convenient to pass 32 bit values in the ESI or EBX register, by all means do so. However, most programmers use the registers above to pass parameters.

If you are passing several parameters to a procedure in the 80x86's registers, you should probably use up the registers in the following order:

First Last

eax, edx, esi, edi, ebx, ecx

In general, you should avoid using EBP register. If you need more than six parameters, perhaps you should pass your values elsewhere.

HLA provides a special high level syntax that lets you tell HLA to pass parameters in one or more of the 80x86 integer registers. Consider the following syntax for an HLA parameter declaration:

```
varname : typename in register
```

In this example, *varname* represents the parameter's name, *typename* is the type of the parameter, and *register* is one of the 80x86's eight-, 16-, or 32-bit integer registers. The size of the data type must be the same as the size of the register (e.g., "int32" is compatible with a 32-bit register). The following is a concrete example of a procedure that passes a character value in a register:

```
procedure swapcase( chToSwap: char in al ); nodisplay; noframe;
begin swapcase;

    if( chToSwap in 'a'..'z' ) then

        and( $5f, chToSwap );   // Convert lower case to upper case.

    elseif( chToSwap in 'A'..'Z' ) then

        or( $20, chToSwap );

    endif;
    ret();

end swapcase;
```

There are a couple of important issues to note here. First, within the procedure's body, the parameter's name is an alias for the corresponding register if you pass the parameter in a register. In other words, *chToSwap* in the previous code is equivalent to "al" (indeed, within the procedure HLA actually defines *chToSwap* as a TEXT constant initialized with the string "al"). Also, since the parameter was passed in a register rather than on the stack, there is no need to build a stack frame for this procedure; hence the absence of the standard entry and exit sequences in the code above. Note that the code above is exactly equivalent to the following code:


```
// Actually, the following parameter list is irrelevant and
// you could remove it.  It does, however, help document the
// fact that this procedure has a single character parameter.

procedure swapcase( chToSwap: char in al ); nodisplay; noframe;
begin swapcase;

    if( al in 'a'..'z' ) then

        and( $5f, al );   // Convert lower case to upper case.

    elseif( al in 'A'..'Z' ) then

        or( $20, al );

    endif;
    ret();

end swapcase;
```

Whenever you call the *swapcase* procedure with some actual (byte sized) parameter, HLA will generate the appropriate code to move that character value into the AL register prior to the call (assuming you don't specify AL as the parameter, in which case HLA doesn't generate any extra code at all). Consider the following calls that the corresponding code that HLA generates:

```
// swapcase( 'a' );

    mov( 'a', al );
    call swapcase;

// swapcase( charVar );

    mov( charVar, al );
    call swapcase;

// swapcase( (type char [ebx]) );

    mov( [ebx], al );
    call swapcase;

// swapcase( ah );

    mov( ah, al );
    call swapcase;

// swapcase( al );

    call swapcase;  // al's value is already in al!
```

The examples above all use the pass by value parameter passing mechanism. When using pass by value to pass parameters in registers, the size of the actual parameter (and formal parameter) must be exactly the same size as the register. Therefore, you are limited to passing eight, sixteen, or thirty-two bit values in the registers by value. Furthermore, these object must be scalar objects. That is, you cannot pass composite (array or record) objects in registers even if such objects are eight, sixteen, or thirty-two bits long.

You can also pass reference parameters in registers. Since pass by reference parameters are four-byte addresses, you must always specify a thirty-two bit register for pass by reference parameters. For example, consider the following *memfill* function that copies a character parameter (passed in AL) throughout some number of memory locations (specified in ECX), at the memory location specified by the value in EDI:

```
// memfill- This procedure stores <ECX> copies of the byte in AL starting
```



```
// at the memory location specified by EDI:

procedure memfill
(
          charVal: char in al;
          count: uns32 in ecx;
    var   dest: byte in edi     // dest is passed by reference
);
    nodisplay; noframe;

begin memfill;

    pushfd();     // Save D flag;
    push( ecx );  // Preserve other registers.
    push( edi );

    cld();        // increment EDI on string operation.
    rep.stosb();  // Store ECX copies of AL starting at EDI.

    pop( edi );
    pop( ecx );
    popfd();
    ret();        // Note that there are no parameters on the stack!

end memfill;
```

It is perfectly possible to pass some parameters in registers and other parameters on the stack to an HLA procedure. Consider the following implementation of *memfill* that passes the *dest* parameter on the stack:

```
procedure memfill
(
          charVal: char in al;
          count: uns32 in ecx;
    var   dest: var
);
    nodisplay;

begin memfill;

    pushfd();          // Save D flag;
    push( ecx );       // Preserve other registers.
    push( edi );

    cld();             // increment EDI on string operation.
    mov( dest, edi );  // get dest address into EDI for STOSB.
    rep.stosb();       // Store ECX copies of AL starting at EDI.

    pop( edi );
    pop( ecx );
    popfd();

end memfill;
```

Of course, you don't have to use the HLA high level procedure calling syntax when passing parameters in the registers. You can manually load the values into registers prior to calling a procedure (with the CALL instruction) and you can refer directly to those values via registers within the procedure. The disadvantage to this scheme, of course, is that the code will be a little more difficult to write, read, and modify. The advantage of the scheme is that you have more control and can pass any eight, sixteen, or thirty-two bit value between the procedure and its callers (e.g., you can load a four-byte array or record into a 32-bit register and call the procedure with that value in a single register, something you cannot do when using the high level language syntax for procedure calls). Fortunately, HLA gives you the choice of whichever parameter pass-

ing scheme is most appropriate, so you can use the manual passing mechanism when it's necessary and use the high level syntax whenever it's not necessary.

There are other parameter passing mechanism beyond pass by value and pass by reference that we will explore in this chapter. We will take a look at ways of passing parameters in registers using those parameter passing mechanisms as we encounter them.

4.3.2 Passing Parameters in FPU and MMX Registers

Since the 80x86's FPU and MMX registers are also registers, it makes perfect sense to pass parameters in these locations if appropriate. Although using the FPU and MMX registers is a little bit more work than using the integer registers, it's generally more efficient than passing the parameters in memory (e.g., on the stack). In this section we'll discuss the techniques and problems associated with passing parameters in these registers.

The first thing to keep in mind is that the MMX and FPU register sets are not independent. These two register sets overlap, much like the eight, sixteen, and thirty-two bit integer registers. Therefore, you cannot pass some parameters in FPU registers and other parameters in MMX registers to a given procedure. For more details on this issue, please see the chapter on the MMX Instruction Set. Also keep in mind that you must execute the EMMS instruction after using the MMX instructions before executing any FPU instructions. Therefore, it's best to partition your code into sections that use the FPU registers and sections that use the MMX registers (or better yet, use only one register set throughout your program).

The FPU represents a fairly special case. First of all, it only makes sense to pass real values through the FPU registers. While it is technically possible to pass other values through the FPU registers, efficiency and accuracy restrictions severely limit what you can do in this regard. This text will not consider passing anything other than real values in the floating point registers, but keep in mind that it is possible to pass generic groups of bits in the FPU registers if you're really careful. Do keep in mind, though, that you need a very detailed knowledge of the FPU if you're going to attempt this (exceptions, rounding, and other issues can cause the FPU to incorrectly manipulate your data under certain circumstances). Needless to say, you can only pass objects by value through the FPU registers; pass by reference isn't applicable here.

Assuming you're willing to pass only real values through the FPU registers, some problems still remain. In particular, the FPU's register architecture does not allow you to load the FPU registers in an arbitrary fashion. Remember, the FPU register set is a stack; so you have to push values onto this stack in the reverse order you wish the values to appear in the register file. For example, if you wish to pass the real variables r, s, and t in FPU registers ST0, ST1, and ST2, you must execute the following code sequence (or something similar):

```
fld( t );    // t -> ST0, but ultimately winds up in ST2.
fld( s );    // s -> ST0, but ultimately winds up in ST1.
fld( r );    // r -> ST0.
```

You cannot load some floating point value into an arbitrary FPU register without a bit of work. Furthermore, once inside the procedure that uses real parameters found on the FPU stack, you cannot easily access arbitrary values in these registers. Remember, FPU arithmetic operations automatically "renumber" the FPU registers as the operations push and pop data on the FPU stack. Therefore, some care and thought must go into the use of FPU registers as parameter locations since those locations are dynamic and change as you manipulate items on the FPU stack.

By far, the most common use of the FPU registers to pass value parameters to a function is to pass a single value parameter in the register so the procedure can operate directly on that parameter via FPU operations. A classic example might be a SIN function that expects its angle in degrees (rather than radians, and the FSIN instruction expects). The function could convert the degree to radians and then execute the FSIN instruction to complete the calculation.

Keep in mind the limited size of the FPU stack. This effectively eliminates the possibility of passing real parameter values through the FPU registers in a recursive procedure. Also keep in mind that it is rather difficult to preserve FPU register values across a procedure call, so be careful about using the FPU registers

to pass parameters since such operations could disturb the values already on the FPU stack (e.g., cause an FPU stack overflow).

The MMX register set, although it shares the same physical silicon as the FPU, does not suffer from the all same problems as the FPU register set when it comes to passing parameters. First of all, the MMX registers are true registers that are individually accessible (i.e., they do not use a stack implementation). You may pass data in any MMX register and you do not have to use the registers in a specific order. Of course, if you pass parameter data in an MMX register, the procedure you're calling must not execute any FPU instructions before you're done with the data or you will lose the value(s) in the MMX register(s).

In theory, you can pass any 64-bit data to a procedure in an MMX register. However, you'll find the use of the MMX register set most convenient if you're actually operating on the data in those registers using MMX instructions.

4.3.3 Passing Parameters in Global Variables

Once you run out of registers, the only other (reasonable) alternative you have is main memory. One of the easiest places to pass parameters is in global variables in the data segment. The following code provides an example:

```
// ThisProc-
//
//   Global variable "Ref1Proc1" contains the address of a pass by reference
//   parameter.  Global variable "Value1Proc1" contains the value of some
//   pass by value parameter.  This procedure stores the value of the
//   "Value1Proc1" parameter into the actual parameter pointed at by
//   "Ref1Proc1".

procedure ThisProc; @nodisplay; @noframe;
begin ThisProc;

    mov( Ref1Proc1, ebx );        // Get address of reference parameter.
    mov( Value1Proc, eax );       // Get Value parameter.
    mov( eax, [ebx] );            // Copy value to actual ref parameter.
    ret();

end ThisProc;
            .
            .
            .

// Sample call to the procedure (includes setting up parameters )

        mov( xxx, eax );           // Pass this parameter by value
        mov( eax, Value1Proc1 );
        lea( eax, yyyy );          // Pass this parameter by reference
        mov( eax, Ref1Proc1 );
        call ThisProc;
```

Passing parameters in global locations is inelegant and inefficient. Furthermore, if you use global variables in this fashion to pass parameters, the subroutines you write cannot use recursion. Fortunately, there are better parameter passing schemes for passing data in memory so you do not need to seriously consider this scheme.

4.3.4 Passing Parameters on the Stack

Most high level languages use the stack to pass parameters because this method is fairly efficient. Indeed, in most of the examples found in this text up to this chapter, passing parameters on the stack has been the standard solution. To pass parameters on the stack, push them immediately before calling the subroutine. The subroutine then reads this data from the stack memory and operates on it appropriately. Consider the following HLA procedure declaration and call:

```
procedure CallProc( a:dword; b:dword; c:dword );
        .
        .
        .

        CallProc(i,j,k+4);
```

By default, HLA pushes its parameters onto the stack in the order that they appear in the parameter list. Therefore, the 80x86 code you would typically write for this subroutine call is[1]

```
push( i );
push( j );
mov( k, eax );
add( 4, eax );
push( eax );
call CallProc;
```

Upon entry into *CallProc*, the 80x86's stack looks like that shown in Figure 4.1

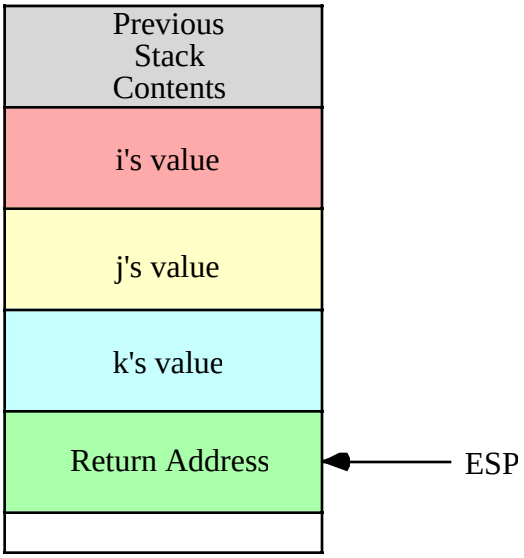

Figure 4.1 Activation Record for CallProc Invocation

Since the chapter on intermediate procedures discusses how to access these parameters, we will not repeat that discussion here. Instead, this section will attempt to tie together material from the previous chapters on procedures and the chapter on Mixed Language Programming.

1. Actually, you'd probably use the HLA high level calling syntax in the typical case, but we'll assume the use of the low-level syntax for the examples appearing in this chapter.

As noted in the chapter on intermediate procedures, the HLA compiler automatically associates some (positive) offset from EBP with each (non-register) parameter you declare in the formal parameter list. Keeping in mind that the base pointer for the activation record (EBP) points at the saved value of EBP and the return address is immediately above that, the first double word of parameter data starts at offset +8 from EBP in the activation record (see Figure 4.2 for one possible arrangement).

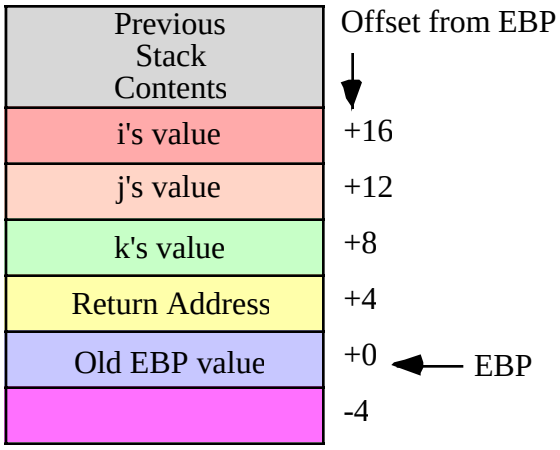

Figure 4.2 Offsets into CallProc's Activation Record

The parameter layout in Figure 4.2 assumes that the caller (as in the previous example) pushes the parameters in the order (left to right) that they appear in the formal parameter list; that is, this arrangement assumes that the code pushes *i* first, *j* second, and *k+4* last. Because this is convenient and easy to do, most high level languages (and HLA, by default) push their parameters in this order. The only problem with this approach is that it winds up locating the first parameter at the highest address in memory and the last parameter at the lowest address in memory. This non-intuitive organization isn't much of a problem because you normally refer to these parameters by their name, not by their offset into the activation record. Hence, whether *i* is at offset +16 or +8 is usually irrelevant to you. Of course, you could refer to these parameters using memory references like "[ebp+16]" or "[ebp+8]" but, in general, that would be exceedingly poor programming style.

In some rare cases, you may actually need to refer to the parameters' values using an addressing mode of the form "[ebp+*disp*]" (where *disp* represents the offset of the parameter into the activation record). One possible reason for doing this is because you've written a macro and that macro always emits a memory operand using this addressing mode. However, even in this case you shouldn't use literal constants like "8" and "16" in the address expression. Instead, you should use the @OFFSET compile-time function to have HLA calculate this offset value for you. I.e., use an address expression of the form:

```
[ebp + @offset( a )]
```

There are two reasons you should specify the addressing mode in this fashion: (1) it's a little more readable this way, and, more importantly, (2) it is easier to maintain. For example, suppose you decide to add a parameter to the end of the parameter list. This causes all the offsets in *CallProc* to change. If you've used address expressions like "[ebp+16]" in you code, you've got to go locate each instance and manually change it. On the other hand, if you use the @OFFSET operator to calculate the offset of the variable in the activation record, then HLA will automatically recompute the current offset of a variable each time you recompile the program; hence you can make changes to the parameter list and not worry about having to manually change the address expressions in your programs.

Although pushing the actual parameters on the stack in the order of the formal parameters' declarations is very common (and the default case that HLA uses), this is not the only order a program can use. Some high level languages (most notably, C, C++, Java, and other C-derived languages) push their parameters in the reverse order, that is, from right to left. The primary reason they do this is to allow variable parameter

lists, a subject we will discuss a little later in this chapter (see “Variable Parameter Lists” on page 1368). Because it is very common for programmers to interface HLA programs with programs written in C, C++, Java, and other such languages, HLA provides a mechanism that allows it to process parameters in this order.

The @CDECL and @STDCALL procedure options tell HLA to reverse the order of the parameters in the activation record. Consider the previous declaration of *CallProc* using the @CDECL procedure option:

```
procedure CallProc( a:dword; b:dword; c:dword ); @cdecl;
        .
        .
        .

        CallProc(i,j,k+4);
```

To implement the call above you would write the following code:

```
mov( k, eax );
add( 4, eax );
push( eax );
push( j );
push( i );
call CallProc;
```

Compare this with the previous version and note that we’ve pushed the parameter values in the opposite order. As a general rule, if you’re not passing parameters between routines written in assembly and C/C++ or you’re not using variable parameter lists, you should use the default parameter passing order (left-to-right). However, if it’s more convenient to do so, don’t be afraid of using the @CDECL or @STDCALL options to reverse the order of the parameters.

Note that using the @CDECL or @STDCALL procedure option immediately changes the offsets of all parameters in a parameter list that has two or more parameters. This is yet another reason for using the @OFFSET operator to calculate the offset of an object rather than manually calculating this. If, for some reason, you need to switch between the two parameter passing schemes, the @OFFSET operator automatically recalculates the offsets.

One common use of assembly language is to write procedures and functions that a high level language program can call. Since different high level languages support different calling mechanisms, you might initially be tempted to write separate procedures for those languages (e.g., Pascal) that push their parameters in a left-to-right order and a separate version of the procedure for those languages (e.g., C) that push their parameters in a right-to-left order. Strictly speaking, this isn’t necessary. You may use HLA’s conditional compilation directives to create a single procedure that you can compile for other high level language. Consider the following procedure declaration fragment:

```
procedure CallProc( a:dword; b:dword; c:dword );
#if( @defined( CLanguage ))
    @cdecl;
#endif
```

With this code, you can compile the procedure for the C language (and similar languages) by simply defining the constant *CLanguage* at the beginning of your code. To compile for Pascal (and similar languages) you would leave the *CLanguage* symbol undefined.

Another issue concerning the use of parameters on the stack is "who takes the responsibility for cleaning these parameters off the stack?" As you saw in the chapter on Mixed Language Programming, various languages assign this responsibility differently. For example, in languages like Pascal, it is the procedure’s responsibility to clean up parameters off the stack before returning control to the caller. In languages like C/C++, it is the caller’s responsibility to clean up parameters on the stack after the procedure returns. By default, HLA procedures use the Pascal calling convention, and therefore the procedures themselves take responsibility for cleaning up the stack. However, if you specify the @CDECL procedure option for a given procedure, then HLA does not emit the code to remove the parameters from the stack when a procedure

returns. Instead, HLA leaves it up to the caller to remove those parameters. Therefore, the call above to CallProc (the one with the @CDECL option) isn't completely correct. Immediately after the call the code should remove the 12 bytes of parameters it has pushed on the stack. It could accomplish this using code like the following:

```
mov( k, eax );
add( 4, eax );
push( eax );
push( j );
push( i );
call CallProc;
add( 12, esp );    // Remove parameters from the stack.
```

Many C compilers don't emit an ADD instruction after each call that has parameters. If there are two or more procedures in a row, and the previous contents of the stack is not needed between the calls, the C compilers may perform a slight optimization and remove the parameter only after the last call in the sequence. E.g., consider the following:

```
pushd( 5 );
call Proc1Parm

push( i );
push( eax );
call Proc2Parms;

add( 12, esp );   // Remove parameters for Proc1Parm and Proc2Parms.
```

The @STDCALL procedure option is a combination of the @CDECL and @PASCAL calling conventions. @STDCALL passes its parameters in the right-to-left order (like C/C++) but requires the procedure to remove the parameters from the stack (like @PASCAL). It's also possible to pass parameters in the left-to-right order (like @PASCAL) and require the caller to remove the parameters from the stack (like C), but HLA does not provide a specific syntax for this. If you want to use this calling convention, you will need to manually build and destroy the activation record, e.g.,

```
procedure CallerPopsParms( i:int32; j:uns32; r:real64 ); nodisplay; noframe;
begin CallerPopsParms;

    push( ebp );
    mov( esp, ebp );
        .
        .
        .
    mov( ebp, esp );
    pop( ebp );
    ret();           // Don't remove any parameters from the stack.

end CallerPopsParms;
        .
        .
        .
    pushd( 5 );
    pushd( 6 );
    pushd( (type dword r[4]));  // Assume r is an eight-byte real.
    pushd( (type dword r));
    call CallerPopsParms;
    add( 16, esp );             // Remove 16 bytes of parameters from stack.
```

Notice how this procedure uses the Pascal calling convention (to get parameters in the left-to-right order) but manually builds and destroys the activation record so that HLA doesn't automatically remove the parameters from the stack. Although the need to operate this way is nearly non-existent, it's interesting to note that it's still possible to do this in assembly language.

4.3.5 Passing Parameters in the Code Stream

The chapter on Intermediate Procedures introduced the mechanism for passing parameters in the code stream with a simple example of a *Print* subroutine. The *Print* routine is a very space-efficient way to print literal string constants to the standard output. A typical call to *Print* takes the following form:

```
call Print
byte "Hello World", 0   // Strings after Print must end with a zero!
```

As you may recall, the *Print* routine pops the return address off the stack and uses this as a pointer to a zero terminated string, printing each character it finds until it encounters a zero byte. Upon finding a zero byte, the *Print* routine pushes the address of the byte following the zero back onto the stack for use as the new return address (so control returns to the instruction following the zero byte). For more information on the *Print* subroutine, see the section on Code Stream Parameters in the chapter on Intermediate Procedures.

The *Print* example demonstrates two important concepts with code stream parameters: passing simple string constants by value and passing a variable length parameter. Contrast this call to *Print* with an equivalent call to the HLA Standard Library *stdout.puts* routine:

```
stdout.puts( "Hello World" );
```

It may look like the call to *stdout.puts* is simpler and more efficient. However, looks can be deceiving and they certainly are in this case. The statement above actually compiles into code similar to the following:

```
    push( HWString );
    call stdout.puts;
        .
        .
        .
// In the CONSTs segment:

          dword  11     // Maximum string length
          dword  11     // Current string length
HWS       byte   "Hello World", 0
HWString  dword  HWS
```

As you can see, the *stdout.puts* version is a little larger because it has three extra dword declarations plus an extra PUSH instruction. (It turns out that *stdout.puts* is faster because it prints the whole string at once rather than a character at a time, but the output operation is so slow anyway that the performance difference is not significant here.) This demonstrates that if you're attempting to save space, passing parameters in the code stream can help.

Note that the *stdout.puts* procedure is more flexible that *Print*. The *Print* procedure only prints string literal constants; you cannot use it to print string variables (as *stdout.puts* can). While it is possible to print string variables with a variant of the *Print* procedure (passing the variable's address in the code stream), this still isn't as flexible as *stdout.puts* because stdout.puts can easily print static and local (automatic) variables whereas this variant of *Print* cannot easily do this. This is why the HLA Standard Library uses the stack to pass the string variable rather than the code stream. Still, it's instructive to look at how you would write such a version of Print, so we'll do that in just a few moments.

One problem with passing parameters in the code stream is that the code stream is read-only[2]. Therefore, any parameter you pass in the code stream must, necessarily, be a constant. While one can easily dream up some functions to whom you always pass constant values in the parameter lists, most procedures work best if you can pass different values (through variables) on each call to a procedure. Unfortunately, this is not possible when passing parameters by value to a procedure through the code stream. Fortunately, we can also pass data by reference through the code stream.

2. Technically, it is possible to make the code segment writable, but we will not consider that possibility here.

When passing reference parameters in the code stream, we must specify the address of the parameter(s) following the CALL instruction in the source file. Since we can only pass constant data (whose value is known at compile time) in the code stream, this means that HLA must know the address of the objects you pass by reference as parameters when it encounters the instruction. This, in turn, means that you will usually pass the address of static objects (STATIC, READONLY, and STORAGE) variables in the code stream. In particular, HLA does not know the address of an automatic (VAR) object at compile time, so you cannot pass the address of a VAR object in the code stream[3].

To pass the address of some static object in the code stream, you would typically use the dword directive and list the object's name in the dword's operand field. Consider the following code that expects three parameters by reference:

Calling sequence:

```
static
    I:uns32;
    J:uns32;
    K:uns32;
        .
        .
        .
    call  AddEm;
    dword I,J,K;
```

Whenever you specify the name of a STATIC object in the operand field of the dword directive, HLA automatically substitutes the four-byte address of that static object for the operand. Therefore, the object code for the instruction above consists of the call to the *AddEm* procedure followed by 12 bytes containing the static addresses of *I*, *J*, and *K*. Assuming that the purpose of this code is to add the values in *J* and *K* together and store the sum into *I*, the following procedure will accomplish this task:

```
procedure AddEm; @nodisplay;
begin AddEm;

    push( eax );          // Preserve the registers we use.
    push( ebx );
    push( ecx );
    mov( [ebp+4], ebx );  // Get the return address.
    mov( [ebx+4], ecx );  // Get J's address.
    mov( [ecx], eax );    // Get J's value.
    mov( [ebx+8], ecx );  // Get K's address.
    add( [ecx], eax );    // Add in K's value.
    mov( [ebx], ecx );    // Get I's address.
    mov( eax, [ecx] );    // Store sum into I.
    add( 12, ebx );       // Skip over addresses in code stream.
    mov( ebx, [ebp+4] );  // Save as new return address.
    pop( ecx );
    pop( ebx );
    pop( eax );

end AddEm;
```

This subroutine adds *J* and *K* together and stores the result into *I*. Note that this code uses 32 bit constant pointers to pass the addresses of *I*, *J*, and *K* to *AddEm*. Therefore, *I*, *J*, and *K* must be in a static data segment. Note at the end of this procedure how the code advances the return address beyond these three pointers in the code stream so that the procedure returns beyond the address of *K* in the code stream.

The important thing to keep in mind when passing parameters in the code stream is that you must always advance the procedure's return address beyond any such parameters before returning from that pro-

3. You may, however, pass the offset of that variable in some activation record. However, implementing the code to access such an object is an exercise that is left to the reader.

cedure. If you fail to do this, the procedure will return into the parameter list and attempt to execute that data as machine instructions. The result is almost always catastrophic. Since HLA does not provide a high level syntax that automatically passes parameters in the code stream for you, you have to manually pass these parameters in your code. This means that you need to be extra careful. For even if you've written your procedure correctly, it's quite possible to create a problem if the calls aren't correct. For example, if you leave off a parameter in the call to *AddEm* or insert an extra parameter on some particular call, the code that adjusts the return address will not be correct and the program will probably not function correctly. So take care when using this parameter passing mechanism.

4.3.6 Passing Parameters via a Parameter Block

Another way to pass parameters in memory is through a *parameter block*. A parameter block is a set of contiguous memory locations containing the parameters. Generally, you would use a record object to hold the parameters. To access such parameters, you would pass the subroutine a pointer to the parameter block. Consider the subroutine from the previous section that adds *J* and *K* together, storing the result in *I*; the code that passes these parameters through a parameter block might be

Calling sequence:

```
type
    AddEmParmBlock:
        record
            i: pointer to uns32;
            j: uns32;
            k: uns32;
        endrecord;

static
    a: uns32;
    ParmBlock: AddEmParmBlock := AddEmParmBlock: [ &a, 2, 3 ];

procedure AddEm( var pb:AddEmParmBlock in esi ); nodisplay;
begin AddEm;

    push( eax );
    push( ebx );
    mov( (type AddEmParmBlock [esi]).j, eax );
    add( (type AddEmParmBlock [esi]).k, eax );
    mov( (type AddEmParmBlock [esi]).i, ebx );
    mov( eax, [ebx] );
    pop( ebx );
    pop( eax );

end AddEm;
```

This form of parameter passing works well when passing several static variables by reference or constant parameters by value, because you can directly initialize the parameter block as was done above.

Note that the pointer to the parameter block is itself a parameter. The examples in this section pass this pointer in a register. However, you can pass this pointer anywhere you would pass any other reference parameter – in registers, in global variables, on the stack, in the code stream, even in another parameter block! Such variations on the theme, however, will be left to your own imagination. As with any parameter, the best place to pass a pointer to a parameter block is in the registers. This text will generally adopt that policy.

Parameter blocks are especially useful when you make several different calls to a procedure and in each instance you pass constant values. Parameter blocks are less useful when you pass variables to procedures, because you will need to copy the current variable's value into the parameter block before the call (this is

roughly equivalent to passing the parameter in a global variable. However, if each particular call to a procedure has a fixed parameter list, and that parameter list contains constants (static addresses or constant values), then using parameter blocks can be a useful mechanism.

Also note that class fields are also an excellent place to pass parameters. Because class fields are very similar to records, we'll not create a separate category for these, but lump class fields together with parameter blocks.

4.4 How You Can Pass Parameters

There are six major mechanisms for passing data to and from a procedure, they are

- pass by value,
- pass by reference,
- pass by value/returned,
- pass by result,
- pass by name, and
- pass by lazy evaluation

Actually, it's quite easy to invent some additional ways to pass parameters beyond these six ways, but this text will concentrate on these particular mechanisms and leave other approaches to the reader to discover.

Since this text has already spent considerable time discussing pass by value and pass by reference, the following subsections will concentrate mainly on the last four ways to pass parameters.

4.4.1 Pass by Value-Result

Pass by value-result (also known as value-returned) combines features from both the pass by value and pass by reference mechanisms. You pass a value-result parameter by address, just like pass by reference parameters. However, upon entry, the procedure makes a temporary copy of this parameter and uses the copy while the procedure is executing. When the procedure finishes, it copies the temporary copy back to the original parameter.

This copy-in and copy-out process takes time and requires extra memory (for the copy of the data as well as the code that copies the data). Therefore, for simple parameter use, pass by value-result may be less efficient than pass by reference. Of course, if the program semantics require pass by value-result, you have no choice but to pay the price for its use.

In some instances, pass by value-returned is more efficient than pass by reference. If a procedure only references the parameter a couple of times, copying the parameter's data is expensive. On the other hand, if the procedure uses this parameter value often, the procedure amortizes the fixed cost of copying the data over many inexpensive accesses to the local copy (versus expensive indirect reference using the pointer to access the data).

HLA supports the use of value/result parameters via the VALRES keyword. If you prefix a parameter declaration with VALRES, HLA will assume you want to pass the parameter by value/result. Whenever you call the procedure, HLA treats the parameter like a pass by reference parameter and generates code to pass the address of the actual parameter to the procedure. Within the procedure, HLA emits code to copy the data referenced by this point to a local copy of the variable[4]. In the body of the procedure, you access the parameter as though it were a pass by value parameter. Finally, before the procedure returns, HLA emits code to copy the local data back to the actual parameter. Here's the syntax for a typical procedure that uses pass by value result:

4. This statement assumes that you're not using the @NOFRAME procedure option.


```
procedure AddandZero( valres p1:uns32;  valres p2:uns32 ); @nodisplay;
begin AddandZero;

    mov( p2, eax );
    add( eax, p1 );
    mov( 0, p2 );

end AddandZero;
```

A typical call to this function might look like the following:

```
    AddandZero( j, k );
```

This call computes "j := j+k;" and "k := 0;" simultaneously.

Note that HLA automatically emits the code within the *AddandZero* procedure to copy the data from *p1* and *p2's* actual parameters into the local variables associated with these parameters. Likewise, HLA emits the code, just before returning, to copy the local parameter data back to the actual parameter. HLA also allocates storage for the local copies of these parameters within the activation record. Indeed, the names *p1* and *p2* in this example are actually associated with these local variables, not the formal parameters themselves. Here's some code similar to that which HLA emits for the *AddandZero* procedure earlier:

```
procedure AddandZero( var p1_ref: uns32; var p2_ref:uns32 );
    @nodisplay;
    @noframe;
var
    p1: uns32;
    p2: uns32;
begin AddandZero;

    push( ebp );
    sub( _vars_, esp );  // Note: _vars_ is "8" in this example.
    push( eax );
    mov( p1_ref, eax );
    mov( [eax], eax );
    mov( eax, p1 );
    mov( p2_ref, eax );
    mov( [eax], eax );
    mov( eax, p2 );
    pop( eax );

    // Actual procedure body begins here:

    mov( p2, eax );
    add( eax, p1 );
    mov( 0, p2 );

    // Clean up code associated with the procedure's return:

    push( eax );
    push( ebx );
    mov( p1_ref, ebx );
    mov( p1, eax );
    mov( eax, [ebx] );
    mov( p2_ref, ebx );
    mov( p2, eax );
    mov( eax, [ebx] );
    pop( ebx );
    pop( eax );
    ret( 8 );

end AddandZero;
```


As you can see from this example, pass by value/result has considerable overhead associated with it in order to copy the data into and out of the procedure's activation record. If efficiency is a concern to you, you should avoid using pass by value/result for parameters you don't reference numerous times within the procedure's body.

If you pass an array, record, or other large data structure via pass by value/result, HLA will emit code that uses a MOVS instruction to copy the data into and out of the procedure's activation record. Although this copy operation will be slow for larger objects, you needn't worry about the compiler emitting a ton of individual MOV instructions to copy a large data structure via value/result.

If you specify the @NOFRAME option when you actually declare a procedure with value/result parameters, HLA does not emit the code to automatically allocate the local storage and copy the actual parameter data into the local storage. Furthermore, since there is no local storage, the formal parameter names refer to the address passed as a parameter rather than to the local storage. For all intents and purposes, specifying @NOFRAME tells HLA to treat the pass by value/result parameters as pass by reference. The calling code passes in the address and it is your responsibility to dereference that address and copy the local data into and out of the procedure. Therefore, it's quite unusual to see an HLA procedure use pass by value/result parameters along with the @NOFRAME option (since using pass by reference achieves the same thing).

This is not to say that you shouldn't use @NOFRAME when you want pass by value/result semantics. The code that HLA generates to copy parameters into and out of a procedure isn't always the most efficient because it always preserves all registers. By using @NOFRAME with pass by value/result parameters, you can supply slightly better code in some instances; however, you could also achieve the same effect (with identical code) by using pass by reference.

When calling a procedure with pass by value/result parameters, HLA pushes the address of the actual parameter on the stack in a manner identical to that for pass by reference parameters. Indeed, when looking at the code HLA generates for a pass by reference or pass by value/result parameter, you will not be able to tell the difference. This means that if you manually want to pass a parameter by value/result to a procedure, you use the same code you would use for a pass by reference parameter; specifically, you would compute the address of the object and push that address onto the stack. Here's code equivalent to what HLA generates for the previous call to *AddandZero*[5]:

```
//  AddandZero( k, j );

        lea( eax, k );
        push( eax );
        lea( eax, j );
        push( eax );
        call AddandZero;
```

Obviously, pass by value/result will modify the value of the actual parameter. Since pass by reference also modifies the value of the actual parameter to a procedure, you may be wondering if there are any semantic differences between these two parameter passing mechanisms. The answer is yes – in some special cases their behavior is different. Consider the following code that is similar to the Pascal code appearing in the chapter on intermediate procedures:

```
procedure uhoh( var  i:int32; var j:int32 ); @nodisplay;
begin uhoh;

    mov( i, ebx );
    mov( 4, (type int32 [ebx]) );
    mov( j, ecx );
    mov( [ebx], eax );
    add( [ecx], eax );
    stdout.put( "i+j=", (type int32 eax), nl );
```

5. Actually, this code is a little more efficient since it doesn't worry about preserving EAX's value; this example assumes the presence of the "@use eax;" procedure option.


```
end uhoh;
    .
    .
    .
var
    k: int32;
    .
    .
    .
    mov( 5, k );
    uhoh( k, k );
    .
    .
    .
```

As you may recall from the chapter on Intermediate Procedures, the call to *uhoh* above prints "8" rather than the expected value of "9". The reason is because *i* and *j* are aliases of one another when you call *uhoh* and pass the same variable in both parameter positions.

If we switch the parameter passing mechanism above to value/result, then *i* and *j* are not exactly aliases of one another so this procedure exhibits different semantics when you pass the same variable in both parameter positions. Consider the following implementation:

```
procedure uhoh( valres  i:int32; valres j:int32 ); nodisplay;
begin uhoh;

    mov( 4, i );
    mov( i, eax );
    add( j, eax );
    stdout.put( "i+j=", (type int32 eax), nl );

end uhoh;
    .
    .
    .
var
    k: int32;
    .
    .
    .
    mov( 5, k );
    uhoh( k, k );
    .
    .
    .
```

In this particular implementation the output value is "9" as you would intuitively expect. The reason this version produces a different result is because *i* and *j* are not aliases of one another within the procedure. These names refer to separate local objects that the procedure happens to initialize with the value of the same variable upon initial entry. However, when the body of the procedure executes, *i* and *j* are distinct so storing four into *i* does not overwrite the value in *j*. Hence, when this code adds the values of *i* and *j* together, *j* still contains the value 5, so this procedure displays the value nine.

Note that there is a question of what value *k* will have when *uhoh* returns to its caller. Since pass by value/result stores the value of the formal parameter back into the actual parameter, the value of *k* could either be four or five (since *k* is the formal parameter associated with both *i* and *j*). Obviously, *k* may only contain one or the other of these values. HLA does not make any guarantees about which value *k* will hold other than it will be one or the other of these two possible values. Obviously, you can figure this out by writing a simple program, but keep in mind that future versions of HLA may not respect the current ordering; worse, it's quite possible that within the same version of HLA, for some calls it could store *i's* value into *k* and for other calls it could store *j's* value into *k* (not likely, but the HLA language allows this). The order by

which HLA copies value/result parameters into and out of a procedure is completely implementation dependent. If you need to guarantee the copying order, then you should use the @NOFRAME option (or use pass by reference) and copy the data yourself.

Of course, this ambiguity exists only if you pass the same actual parameter in two value/result parameter positions on the same call. If you pass different actual variables, this problem does not exist. Since it is very rare for a program to pass the same variable in two parameter slots, particularly two pass by value/result slots, it is unlikely you will ever encounter this problem.

HLA implements pass by value/result via pass by reference and copying. It is also possible to implement pass by value/result using pass by value and copying. When using the pass by reference mechanism to support pass by value/result, it is the procedure's responsibility to copy the data from the actual parameter into the local copy; when using the pass by value form, it is the caller's responsibility to copy the data to and from the local object. Consider the following implementation that (manually) copies the data on the call and return from the procedure:

```
procedure DisplayAndClear( val i:int32 ); @nodisplay; @noframe;
begin DisplayAndClear;

    push( ebp );       // NOFRAME, so we have to do this manually.
    mov( esp, ebp );

    stdout.put( "I = ", i, nl );
    mov( 0, i );

    pop( ebp );
    ret();             // Note that we don't clean up the parameters.

end DisplayAndClear;
    .
    .
    .
push( m );
call DisplayAndClear;
pop( m );
stdout.put( "m = ", m, nl );
    .
    .
    .
```

The sequence above displays "I = 5" and "m = 0" when this code sequence runs. Note how this code passes the value in on the stack and then returns the result back on the stack (and the caller copies the data back to the actual parameter.

In the example above, the procedure uses the @NOFRAME option in order to prevent HLA from automatically removing the parameter data from the stack. Another way to achieve this effect is to use the @CDECL procedure option (that tells HLA to use the C calling convention, which also leaves the parameters on the stack for the caller to clean up). Using this option, we could rewrite the code sequence above as follows:

```
procedure DisplayAndClear( val i:int32 ); @nodisplay; @cdecl;
begin DisplayAndClear;

    stdout.put( "I = ", i, nl );
    mov( 0, i );

end DisplayAndClear;
    .
    .
    .
DisplayAndClear( m );
pop( m );
```



```
    stdout.put( "m = ", m, nl );
        .
        .
        .
```

The advantage to this scheme is that HLA automatically emits the procedure's entry and exit sequences so you don't have to manually supply this information. Keep in mind, however, that the @CDECL calling sequence pushes the parameters on the stack in the reverse order of the standard HLA calling sequence. Generally, this won't make a difference to you code unless you explicitly assume the order of parameters in memory. Obviously, this won't make a difference at all when you've only got a single parameter.

The examples in this section have all assumed that we've passed the value/result parameters on the stack. Indeed, HLA only supports this location if you want to use a high level calling syntax for value/result parameters. On the other hand, if you're willing to manually pass the parameters in and out of a procedure, then you may pass the value/result parameters in other locations including the registers, in the code stream, in global variables, or in parameter blocks.

Passing parameters by value/result in registers is probably the easiest way to go. All you've got to do is load an appropriate register with the desired value before calling the procedure and then leave the return value in that register upon return. When the procedure returns, it can use the register's value however it sees fit. If you prefer to pass the value/result parameter by reference rather than by value, you can always pass in the address of the actual object in a 32-bit register and do the necessary copying within the procedure's body.

Of course, there are a couple of drawbacks to passing value/result parameters in the registers; first, the registers can only hold small, scalar, objects (though you can pass the address of a large object in a register). Second, there are a limited number of registers. But if you can live these drawbacks, registers provide a very efficient place to pass value/result parameters.

It is possible to pass certain value/result parameters in the code stream. However, you'll always pass such parameters by their address (rather than by value) to the procedure since the code stream is in read-only memory (and you can't write a value back to the code stream). When passing the actual parameters via value/result, you must pass in the address of the object in the code stream, so the objects must be static variables so HLA can compute their addresses at compile-time. The actual implementation of value/result parameters in the code stream is left as an exercise for the end of this volume.

There is one advantage to value/result parameters in the HLA/assembly programming environment. You get semantics very similar to pass by reference without having to worry about constant dereferencing of the parameter throughout the code. That is, you get the ability to modify the actual parameter you pass into a procedure, yet within the procedure you get to access the parameter like a local variable or value parameter. This simplification makes it easier to write code and can be a real time saver if you're willing to (sometimes) trade off a minor amount of performance for easier to read-and-write code.

4.4.2 Pass by Result

Pass by result is almost identical to pass by value-result. You pass in a pointer to the desired object and the procedure uses a local copy of the variable and then stores the result through the pointer when returning. The only difference between pass by value-result and pass by result is that when passing parameters by result you do not copy the data upon entering the procedure. Pass by result parameters are for returning values, not passing data to the procedure. Therefore, pass by result is slightly more efficient than pass by value-result since you save the cost of copying the data into the local variable.

HLA supports pass by result parameters using the RESULT keyword prior to a formal parameter declaration. Consider the following procedure declaration:

```
procedure HasResParm( result r:uns32 ); nodisplay;
begin HasResParm;

    mov( 5, r );

end HasResParm;
```


Like pass by value/result, modification of the pass by result parameter results (ultimately) in the modification of the actual parameter. The difference between the two parameter passing mechanisms is that pass by result parameters do not have a known initial value upon entry into the code (i.e., the HLA compiler does not emit code to copy any data into the parameter upon entry to the procedure).

Also like pass by value/result, you may pass result parameters in locations other than on the stack. HLA does not support anything other than the stack when using the high level calling syntax, but you may certainly pass result parameters manually in registers, in the code stream, in global variables, and in parameter blocks.

4.4.3 Pass by Name

Some high level languages, like ALGOL-68 and Panacea, support pass by name parameters. Pass by name produces semantics that are similar (though not identical) to textual substitution (e.g., like macro parameters). However, implementing pass by name using textual substitution in a compiled language (like ALGOL-68) is very difficult and inefficient. Basically, you would have to recompile a function every time you call it. So compiled languages that support pass by name parameters generally use a different technique to pass those parameters. Consider the following Panacea procedure (Panacea's syntax is sufficiently similar to HLA's that you should be able to figure out what's going on):

```
PassByName: procedure(name item:integer; var index:integer);
begin PassByName;

    foreach index in 0..10 do

        item := 0;

    endfor;

end PassByName;
```

Assume you call this routine with the statement "PassByName(A[i], i);" where A is an array of integers having (at least) the elements *A[0]..A[10]*. Were you to substitute (textually) the pass by name parameter *item* you would obtain the following code:

```
begin PassByName;

    foreach I in 0..10 do

        A[I] := 0;

    endfor;

end PassByName;
```

This code zeros out elements 0..10 of array *A*.

High level languages like ALGOL-68 and Panacea compile pass by name parameters into *functions* that return the address of a given parameter. So in one respect, pass by name parameters are similar to pass by reference parameters insofar as you pass the address of an object. The major difference is that with pass by reference you compute the address of an object before calling a subroutine; with pass by name the subroutine itself calls some function to compute the address of the parameter whenever the function references that parameter.

So what difference does this make? Well, reconsider the code above. Had you passed *A[I]* by reference rather than by name, the calling code would compute the address of *A[I]* just before the call and passed in this address. Inside the *PassByName* procedure the variable item would have always referred to a single address, not an address that changes along with *I*. With pass by name parameters, *item* is really a function

that computes the address of the parameter into which the procedure stores the value zero. Such a function might look like the following:

```
procedure ItemThunk; @nodisplay; @noframe;
begin ItemThunk;

    mov( i, eax );
    lea( eax, A[eax*4] );
    ret();

end ItemThunk;
```

The compiled code inside the *PassByName* procedure might look something like the following:

```
; item := 0;

    call ItemThunk;
    mov( 0, (type dword [eax]));
```

Thunk is the historical term for these functions that compute the address of a pass by name parameter. It is worth noting that most HLLs supporting pass by name parameters do not call thunks directly (like the call above). Generally, the caller passes the address of a thunk and the subroutine calls the thunk indirectly. This allows the same sequence of instructions to call several different thunks (corresponding to different calls to the subroutine). In HLA, of course, we will use HLA thunk variables for this purpose. Indeed, when you declare a procedure with a pass by name parameter, HLA associates the thunk type with that parameter. The only difference between a parameter whose type is thunk and a pass by name parameter is that HLA requires a thunk constant for the pass by name parameter (whereas a parameter whose type is thunk can be either a thunk constant or a thunk variable). Here's a typical procedure prototype using a pass by name variable (note the use of the NAME keyword to specify pass by name):

```
procedure HasNameParm( name nameVar:uns32 );
```

Since *nameVar* is a thunk, you call this object rather than treat it as data or as a pointer. Although HLA doesn't enforce this, the convention is that a pass by name parameter returns the address of the object whenever you invoke the thunk. The procedure then dereferences this address to access the actual data. The following code is the HLA equivalent of the Panacea procedure given earlier:

```
procedure passByName( name ary:int32; var ip:int32 ); @nodisplay;
const i:text := "(type int32 [ebx])";
begin passByName;

    mov( ip, ebx );
    mov( 0, i );
    while( i <= 10 ) do

        ary();   // Get address of "ary[i]" into eax.
        mov(i, ecx );
        mov( ecx, (type int32 [eax]) );
        inc( i );

    endwhile;

end passByName;
```

Notice how this code assumes that the *ary* thunk returns a pointer in the EAX register.

Whenever you call a procedure with a pass by name parameter, you must supply a thunk that computes some address and returns that address in the EAX register (or wherever you expect the address to be sitting upon return from the thunk; convention dictates the EAX register). Here is some same code that demonstrates how to pass a thunk constant for a pass by name parameter to the procedure above:

```
var
```



```
index:uns32;
array: uns32[ 11 ];  // Has elements 0..10.
    .
    .
    .
passByName
(
    thunk
    #{
        push( ebx );
        mov( index, ebx );
        lea( eax, array[ebx*4] );
        pop( ebx );
    }#,
    index
);
```

The "thunk #{...}#" sequence specifies a literal thunk that HLA compiles into the code stream. For the environment pointer, HLA pushes the current value for EBP, for the procedure pointer, HLA passes in the address of the code in the "#{...}#" braces. Whenever the *passByName* procedure actually calls this thunk, the run-time system restores EBP with the pointer to the current procedure's activation record and executes the code in these braces. If you look carefully at the code above, you'll see that this code loads the EAX register with the address of the *array[index]* variable. Therefore, the *passByName* procedure will store the next value into this element of *array.*

Pass by name parameter passing has garnered a bad name because it is a notoriously slow mechanism. Instead of directly or indirectly accessing an object, you have to first make a procedure call (which is expensive compared to an access) and then dereference a pointer. However, because pass by name parameters defer their evaluation until you actually access an object, pass by name effectively gives you a deferred pass by reference parameter passing mechanism (deferring the calculation of the address of the parameter until you actually access that parameter). This can be very important in certain situations. As you've seen in the chapter on thunks, the proper use of deferred evaluation can actually improve program performance. Most of the complaints about pass by name are because someone misused this parameter passing mechanism when some other mechanism would have been more appropriate. There are times, however, when pass by name is the best approach.

It is possible to transmit pass by name parameters in some location other than the stack. However, we don't call them pass by name parameters anymore; they're just thunks (that happen to return an address in EAX) at that point. So if you wish to pass a pass by name parameter in some other location than the stack, simply create a thunk object and pass your parameter as the thunk.

4.4.4 Pass by Lazy-Evaluation

Pass by name is similar to pass by reference insofar as the procedure accesses the parameter using the address of the parameter. The primary difference between the two is that a caller directly passes the address on the stack when passing by reference, it passes the address of a function that computes the parameter's address when passing a parameter by name. The pass by lazy evaluation mechanism shares this same relationship with pass by value parameters – the caller passes the address of a function that computes the parameter's value if the first access to that parameter is a read operation.

Pass by lazy evaluation is a useful parameter passing technique if the cost of computing the parameter value is very high and the procedure may not use the value. Consider the following HLA procedure header:

```
procedure PassByEval( lazy a:int32; lazy b:int32; lazy c:int32 );
```

Consider the *PassByEval* procedure above. Suppose it takes several minutes to compute the values for the *a*, *b*, and *c* parameters (these could be, for example, three different possible paths in a Chess game). Perhaps the *PassByEval* procedure only uses the value of one of these parameters. Without pass by lazy evaluation, the calling code would have to spend the time to compute all three parameters even though the

procedure will only use one of the values. With pass by lazy evaluation, however, the procedure will only spend the time computing the value of the one parameter it needs. Lazy evaluation is a common technique artificial intelligence (AI) and operating systems use to improve performance since it provides deferred parameter evaluation capability.

HLA's implementation of pass by lazy evaluation parameters is (currently) identical to the implementation of pass by name parameters. Specifically, pass by lazy evaluation parameters are thunks that you must call within the body of the procedure and that you must write whenever you call the procedure. The difference between pass by name and pass by lazy evaluation is the convention surrounding what the thunks return. By convention, pass by name parameters return a pointer in the EAX register. Pass by lazy evaluation parameters, on the other hand, return a value, not an address. Where the pass by lazy evaluation thunk returns its value depends upon the size of the value. However, by convention most programmers return eight, 16-, 32-, and 64-bit values in the AL, AX, EAX, and EDX:EAX registers, respectively. The exceptions are floating point values (the convention is to use the ST0 register) and MMX values (the convention is to use the MM0 register for MMX values).

Like pass by name, you only pass by lazy evaluation parameters on the stack. Use thunks if you want to pass lazy evaluation parameters in a different location.

Of course, nothing is stopping you from returning a value via a pass by name thunk or an address via a pass by lazy evaluation thunk, but to do so is exceedingly poor programming style. Use these parameter pass mechanisms as they were intended.

4.5 Passing Parameters as Parameters to Another Procedure

When a procedure passes one of its own parameters as a parameter to another procedure, certain problems develop that do not exist when passing variables as parameters. Indeed, in some (rare) cases it is not logically possible to pass some parameter types to some other procedure. This section deals with the problems of passing one procedure's parameters to another procedure.

Pass by value parameters are essentially no different than local variables. All the techniques in the previous sections apply to pass by value parameters. The following sections deal with the cases where the calling procedure is passing a parameter passed to it by reference, value-result, result, name, and lazy evaluation.

4.5.1 Passing Reference Parameters to Other Procedures

Passing a reference parameter though to another procedure is where the complexity begins. Consider the following HLA procedure skeleton:

```
procedure ToProc(???? parm:dword);
begin ToProc;
    .
    .
    .
end ToProc;

procedure HasRef(var refparm:dword);

begin HasRef;
     .
     .
     .
    ToProc(refParm);
     .
     .
```



```
    .
end HasRef;
```

The "????" in the *ToProc* parameter list indicates that we will fill in the appropriate parameter passing mechanism as the discussion warrants.

If *ToProc* expects a pass by value parameter (i.e., ???? is just an empty string), then *HasRef* needs to fetch the value of the *refparm* parameter and pass this value to *ToProc*. The following code accomplishes this[6]:

```
mov( refparm, ebx );  // Fetch address of actual refparm value.
pushd( [ebx] );       // Pass value of refparm variable on the stack.
call ToProc;
```

To pass a reference parameter by reference, value-result, or result parameter is easy – just copy the caller's parameter as-is onto the stack. That is, if the *parm* parameter in *ToProc* above is a reference parameter, a value-result parameter, or a result parameter, you would use the following calling sequence:

```
push( refparm );
call ToProc;
```

We get away with passing the value of *refparm* on the stack because *refparm* currently contains the address of the actual object that *ToProc* will reference. Therefore, we need only copy this value (which is an address).

To pass a reference parameter by name is fairly easy. Just write a thunk that grabs the reference parameter's address and returns this value. In the example above, the call to *ToProc* might look like the following:

```
ToProc
(
    thunk
    #{
        mov( refparm, eax );
    }#
);
```

To pass a reference parameter by lazy evaluation is very similar to passing it by name. The only difference (in *ToProc's* calling sequence) is that the thunk must return the value of the variable rather than its address. You can easily accomplish this with the following thunk:

```
ToProc
(
    thunk
    #{
        mov( refparm, eax );  // Get the address of the actual parameter
        mov( [eax], eax );    // Get the value of the actual parameter.
    }#
);
```

Note that HLA's high level procedure calling syntax automatically handles passing reference parameters as value, reference, value/result, and result parameters. That is, when using the high level procedure call syntax and *ToProc's* parameter is pass by value, pass by reference, pass by value/result, or pass by result, you'd use the following syntax to call *ToProc*:

```
ToProc( refparm );
```

HLA will automatically figure out what data it needs to push on the stack for this procedure call.

---

6. The examples in this section all assume the use of a display. If you are using static links, be sure to adjust all the offsets and the code to allow for the static link that the caller must push immediately before a call.

Unfortunately, HLA does not automatically handle the case where you pass a reference parameter to a procedure via pass by name or pass by lazy evaluation. You must explicitly write this thunk yourself.

4.5.2 Passing Value-Result and Result Parameters as Parameters

If you have a pass by value/result or pass by result parameter that you want to pass to another procedure, and you use the standard HLA mechanism for pass by value/result or pass by result parameters, passing those parameters on to another procedure is trivial because HLA creates local variables using the parameters' names. Therefore, there is no difference between a local variable and a pass by value/result or pass by result parameter in this particular case. Once HLA has made a local copy of the value-result or result parameter or allocates storage for it, you can treat that variable just like a value parameter or a local variable with respect to passing it on to other procedures. In particular, if you're using the HLA high level calling syntax in your code, HLA will automatically pass that procedure by value, reference, value/result, or by result to another procedure. If you're passing the parameter by name or by lazy evaluation to another procedure, you must manually write the thunk that computes the address of the variable (pass by name) or obtains the value of the variable (pass by lazy evaluation) prior to calling the other procedure.

Of course, it doesn't make sense to use the value of a result parameter until you've stored a value into that parameter's local storage. Therefore, take care when passing result parameters to other procedures that you've initialized a result parameter before using its value.

If you're manually passing pass by value/result or pass by result parameters to a procedure and then you need to pass those parameters on to another procedure, HLA cannot automatically generate the appropriate code to pass those parameters on. This is especially true if you've got the parameter sitting in a register or in some location other than a local variable. Likewise, if the procedure you're calling expects you to pass a value/result or result parameter using some mechanism other than passing the address on the stack, you will also have manually write the code to pass the parameter on through. Since such situations are specific to a given situation, the only advice this text can offer is to suggest that you carefully think through what you're doing. Remember, too, that if you use the @NOFRAME procedure option, HLA does not make local copies, so you will have to compute and pass the addresses of such parameters manually.

4.5.3 Passing Name Parameters to Other Procedures

Since a pass by name parameter's thunk returns the address of a parameter, passing a name parameter to another procedure is very similar to passing a reference parameter to another procedure. The primary differences occur when passing the parameter on as a name parameter.

Unfortunately, HLA's high level calling syntax doesn't automatically deal with pass by name parameters. The reason is because HLA doesn't assume that thunks return an address in the EAX register (this is a convention, not a requirement). A programmer who writes the thunk could return the address somewhere else, or could even create a thunk that doesn't return an address at all! Therefore, it is up to you to handle passing pass by name parameters to other procedures.

When passing a name parameter as a value parameter to another procedure, you first call the thunk, dereference the address the thunk returns, and then pass the value to the new procedure. The following code demonstrates such a call when the thunk returns the variable's address in EAX:

```
CallThunk();     // Call the thunk which returns an address in EAX.
pushd( [eax] );  // Push the value of the object as a parameter.
call ToProc;     // Call the procedure that expects a value parameter.
 .
 .
 .
```

Passing a name parameter to another procedure by reference is very easy. All you have to do is push the address the thunk returns onto the stack. The following code, that is very similar to the code above, accomplishes this:


```
CallThunk();     // Call the thunk which returns an address in EAX.
push( eax );     // Push the address of the object as a parameter.
call ToProc;     // Call the procedure that expects a value parameter.
 .
 .
 .
```

Passing a name parameter to another procedure as a pass by name parameter is very easy; all you need to do is pass the thunk on to the new procedure. The following code accomplishes this:

```
push( (type dword CallThunk));
push( (type dword CallThunk[4]));
call ToProc;
```

To pass a name parameter to another procedure by lazy evaluation, you need to create a thunk for the lazy-evaluation parameter that calls the pass by name parameter's thunk, dereferences the pointer, and then returns this value. The implementation is left as a programming project.

4.5.4 Passing Lazy Evaluation Parameters as Parameters

Lazy evaluation are very similar to name parameters except they typically return a value in EAX (or some other register) rather than an address. This means that you may only pass lazy evaluation parameters by value or by lazy evaluation to another procedure (since they don't have an address associated with them).

4.5.5 Parameter Passing Summary

The following table describes how to pass parameters from one procedure as parameters to another procedure. The rows specify the "input" parameter passing mechanism (how the parameter was passed into the current procedure) and the rows specify the "output" parameter passing mechanism (how the procedure passing the parameter on to another procedure as a parameter).

Table 1: Passing Parameters as Parameters to Another Procedure

	Pass as Value	Pass as Reference	Pass as Value-Result	Pass as Result	Pass as Name	Pass as Lazy Evaluation
Value	Pass the value	Pass address of the value parameter	Pass address of the value parameter	Pass address of the value parameter	Create a thunk that returns the address of the value parameter	Create a thunk that returns the value
Reference	Dereference parameter and pass the value it points at	Pass the address (value of the reference parameter)	Pass the address (value of the reference parameter)	Pass the address (value of the reference parameter)	Create a thunk that passes the address (value of the reference parameter)	Create a thunk that deferences the reference parameter and returns its value

Table 1: Passing Parameters as Parameters to Another Procedure

	Pass as Value	Pass as Reference	Pass as Value-Result	Pass as Result	Pass as Name	Pass as Lazy Evaluation
Value-Result	Pass the local value as the value parameter	Pass the address of the local value as the parameter	Pass the address of the local value as the parameter	Pass the address of the local value as the parameter	Create a thunk that returns the address of the local value of the value-result parameter	Create a thunk that returns the value in the local value of the value-result parameter
Result	Pass the local value as the value parameter	Pass the address of the local value as the parameter	Pass the address of the local value as the parameter	Pass the address of the local value as the parameter	Create a thunk that returns the address of the local value of the result parameter	Create a thunk that returns the value in the local value of the result parameter
Name	Call the thunk, dereference the pointer, and pass the value at the address the thunk returns	Call the thunk and pass the address it returns as the parameter	Call the thunk and pass the address it returns as the parameter	Call the thunk and pass the address it returns as the parameter	Pass the address of the thunk and any other values associated with the name parameter	Write a thunk that calls the name parameter's thunk, dereferences the address it returns, and then returns the value at that address
Lazy Evaluation	If necessary, call the thunk to obtain the Lazy Eval parameter's value. Pass the local value as the value parameter	Not possible. Lazy Eval parameters return a value which does not have an address.	Not possible. Lazy Eval parameters return a value which does not have an address.	Not possible. Lazy Eval parameters return a value which does not have an address.	Not possible. Lazy Eval parameters return a value which does not have an address.	Create a thunk that calls the caller's Lazy Eval parameter. This new thunk returns that result as its result.

4.6 Variable Parameter Lists

On occasion you may need the ability to pass a varying number of parameters to a given procedure. The *stdout.put* routine in the HLA Standard Library provides a good example of where having a variable number of parameters is useful. There are two ways to accomplish this: (1) Fake it and use a macro rather than a procedure (this is what the HLA Standard Library does, for example, with the *stdout.put* invocation – *stdout.put* is a macro not a procedure), and (2) Pass in some information to the procedure the describes how many parameters in must process and where it can find those parameters. We'll take a look at both of these mechanisms in this section.

HLA's macro facility allows a varying number of parameters by specifying an empty array parameter as the last formal parameter in the macro list, e.g.,

```
#macro VariableNumOfParms( a, b, c[] );
    .
    .
    .
#endmacro;
```

Whenever HLA processes a macro declaration like the one above, it associates the first two actual parameters with the formal parameters *a* and *b*; any remaining actual parameters becomes strings in the constant string array *c*. By using the @ELEMENTS compile-time function, you can determine how many additional parameters appear in the parameter list (which can be zero or more).

Of course, a macro is not a procedure. So the fact that we have a list of text constants and a string array that represents our actual parameter list does not satisfy the requirements for a varying parameter list at run-time. However, we can write some compile-time code that parses the parameter list and calls an appropriate set of procedures to handle each and every parameter passed to the macro. For example, the *stdout.put* macro splits up the parameter list and calls a sequence of routines (like *stdout.puts* and *stdout.puti32*) to handle each parameter individually.

Breaking up a macro's variable parameter list into a sequence of procedure calls with only one parameter per call may not solve a need you have for varying parameter lists. That being the case, it may still be possible to use macros to implement varying parameters for a procedure. If the number of parameters is within some range, then you can use the function overloading trick discussed in the chapter on macros to call one of several different procedures, each with a different number of parameters. Please see the chapter on macros for additional details.

Although macros provide a convenient way to implement a varying parameter list, they do suffer from some limitations that make them unsuitable for all applications. In particular, if you need to call a single procedure and pass it an indeterminate number of parameters (no limits), then the tricks with macros employed above won't work well for you. In this situation you will need to push the parameters on the stack (or pass them somewhere else) and include some information that tells the procedure how many parameters you're passing (and, perhaps, their size). The most common way to do this is to push the parameters onto the stack and then, as the last parameter, push the parameter count (or size) onto the stack.

Most procedures that push a varying number of parameters on the stack use the C/C++ calling convention. There are two reasons for this: (1) the parameters appear in memory in a natural order (the number of parameters followed by the first parameter, followed by the second parameter, etc.), (2) the caller will need to remove the parameters from the stack since each call can have a different number of parameters and the 80x86 RET instruction can only remove a fixed (constant) number of parameter bytes.

One drawback to using the C/C++ calling convention to pass a variable number of parameters on the stack is that you must manually push the parameters and issue a CALL instruction; HLA does not provide a high-level language syntax for declaring and calling procedures with a varying number of parameters.

Consider, as an example, a simple *MaxUns32* procedure that computes the maximum of *n* uns32 values passed to it. The caller begins by pushing *n* uns32 values and then, finally, it also pushes *n*. Upon return, the caller removes the *n*+1 uns32 values (including *n*) from the stack. The function, presumably, returns the maximum value in the EAX register. Here's a typical call to *MaxUns32*:

```
push( i );
push( j );
push( k );
pushd( 10 );

push( 4 );       // n=4, number of parameters passed to this code.
call MaxUns32;   // Compute the maximum of the above.
add( 20, esp );  // Remove the parameters from the stack.
```

The *MaxUns32* procedure itself must first fetch the value for *n* from a known location (in this case, it will be just above the return address on the stack). The procedure can then use this value to step through each of the other parameters found on the stack above the value for *n*. Here's some sample code that accomplishes this:

```
procedure MaxUns32; nodisplay; noframe;
const n:text := "(type uns32 [ebp+8])";
const first:text := "(type uns32 [ebp+12])";
begin MaxUns32;

    push( ebp );
    mov( esp, ebp );
    push( ebx );
    push( ecx );

    mov( n, ecx );
    if( ecx > 0 ) then

        lea( ebx, first );
        mov( first, eax );     // Use this as the starting Max value.
        repeat

            if( eax < [ebx] ) then

                mov( [ebx], eax );

            endif;
            add( 4, ebx );
            dec( ecx );

        until( ecx = 0 );

    else

        // There were no parameter values to try, so just return zero.

        xor( eax, eax );

    endif;
    pop( ecx );
    pop( ebx );
    pop( ebp );
    ret();        // Can't remove the parameters!

end MaxUns32;
```


This code assumes that *n* is at location [ebp+8] (which it will be if n is the last parameter pushed onto the stack) and that *n* uns32 values appear on the stack above this point. It steps through each of these values searching for the maximum, which the function returns in EAX. If n contains zero upon entry, this function simply returns zero in EAX.

Passing a single parameter count, as above, works fine if all the parameters are the same type and size. If the size and/or type of each parameter varies, you will need to pass information about each individual parameter on the stack. There are many ways to do this, a typical mechanism is to simply preface each parameter on the stack with a double word containing its size in bytes. Another solution is that employed by the *printf* function in the C standard library - pass an array of data (a string in the case of *printf*) that contains type information that the procedure can interpret at run-time to determine the type and size of the parameters. For example, the C *printf* function uses format strings like "%4d" to determine the size (and count, via the number of formatting options that appear within the string) of the parameters.

4.7 Function Results

Functions return a result, which is nothing more than a result parameter. In assembly language, there are very few differences between a procedure and a function. That is why there isn't a "function" directive. Functions and procedures are usually different in high level languages, function calls appear only in expressions, subroutine calls as statements[7]. Assembly language doesn't distinguish between them.

You can return function results in the same places you pass and return parameters. Typically, however, a function returns only a single value (or single data structure) as the function result. The methods and locations used to return function results is the subject of the next four sections.

4.7.1 Returning Function Results in a Register

Like parameters, the 80x86's registers are the best place to return function results. The *getc* routine in the HLA Standard Library is a good example of a function that returns a value in one of the CPU's registers. It reads a character from the keyboard and returns the ASCII code for that character in the AL register. By convention, most programmers return function results in the following registers:

Use	First	Last
Bytes:	al, ah, dl, dh, cl, ch, bl, bh	
Words:	ax, dx, cx, si, di, bx	
Double words:	eax, edx, ecx, esi, edi, ebx	
Quad words:	edx:eax	
Real Values:	ST0	
MMX Values:	MM0	

Once again, this table represents general guidelines. If you're so inclined, you could return a double word value in (CL, DH, AL, BH). If you're returning a function result in some registers, you shouldn't save and restore those registers. Doing so would defeat the whole purpose of the function.

7. "C" is an exception to this rule. C's procedures and functions are all called functions. PL/I is another exception. In PL/I, they're all called procedures.

4.7.2 Returning Function Results on the Stack

Another good place where you can return function results is on the stack. The idea here is to push some dummy values onto the stack to create space for the function result. The function, before leaving, stores its result into this location. When the function returns to the caller, it pops everything off the stack except this function result. Many HLLs use this technique (although most HLLs on the IBM PC return function results in the registers). The following code sequences show how values can be returned on the stack:

```
procedure RtnOnStack( RtnResult: dword;  parm1: uns32; parm2:uns32 );
    @nodisplay;
    @noframe;
var
    LocalVar: uns32;
begin RtnOnStack;

    push( ebp );          // The standard entry sequence
    mov( esp, ebp );
    sub( _vars_, esp );

        << code that leaves a value in RtnResult >>

    mov( ebp, esp );      // Not quite standard exit sequence.
    pop( ebp );
    ret( __parms_-4 );    // Don't pop RtnResult off stack on return!

end RtnOnStack;
```

Calling sequence:

```
    RtnOnStack( 0, p1, p2 );  // "0" is a dummy value to reserve space.
    pop( eax );               // Retrieve return result from stack.
```

Although the caller pushed 12 bytes of data onto the stack, *RtnOnStack* only removes eight bytes. The first "parameter" on the stack is the function result. The function must leave this value on the stack when it returns.

4.7.3 Returning Function Results in Memory Locations

Another reasonable place to return function results is in a known memory location. You can return function values in global variables or you can return a pointer (presumably in a register or a register pair) to a parameter block. This process is virtually identical to passing parameters to a procedure or function in global variables or via a parameter block.

Returning parameters via a pointer to a parameter block is an excellent way to return large data structures as function results. If a function returns an entire array, the best way to return this array is to allocate some storage, store the data into this area, and leave it up to the calling routine to deallocate the storage. Most high level languages that allow you to return large data structures as function results use this technique.

Of course, there is very little difference between returning a function result in memory and the pass by result parameter passing mechanism. See "Pass by Result" on page 1359 for more details.

4.7.4 Returning Large Function Results

Returning small scalar values in the registers or on the stack makes a lot of sense. However, mechanism for returning function results does not scale very well to large data structures. The registers are too small to

return large records or arrays and returning such data on the stack is a lot of work (not to mention that you've got to copy the data from the stack to it's final resting spot upon return from the function). In this section we'll take a look at a couple of methods for returning large objects from a function.

The traditional way to return a large function result is to pass the location where one is to store the result as a pass by reference parameter. The advantage to this scheme is that it is relatively efficient (speed-wise) and doesn't require any extra space; the procedure uses the final destination location as scratch pad memory while it is building up the result. The disadvantage to this scheme is that it is very common to pass the destination variable as an input parameter (thus creating an alias). Since, in a high level language, you don't have the problems of aliases with function return results, this is a non-intuitive semantic result that can create some unexpected problems.

A second solution, though a little bit less efficient, is to use a pass by result parameter to return the function result. Pass by result parameters get their own local copy of the data that the system copies back over the destination location once the function is complete (thus avoiding the problem with aliases). The drawback to using pass by result, especially with large return values, is the fact that the program must copy the data from the local storage to the destination variable when the function completes. This data copy operation can take a significant amount of time for really large objects.

Another solution for returning large objects, that is relatively efficient, is to allocate storage for the object in the function, place whatever data you wish to return in the allocated storage, and then return a pointer to this storage. If the calling code references this data indirectly rather than copying the data to a different location upon return, this mechanism and run significantly faster than pass by result. Of course, it is not as general as using pass by result parameters, but with a little planning it is easy to arrange you code so that it works with pointers to large objects. String functions are probably the best example of this function result return mechanism in practice. It is very common for a function to allocate storage for a string result on the heap and then return a "string variable" in EAX (remember that strings in HLA are pointers).

4.8 Putting It All Together

This chapter discusses how and where you can pass parameters in an assembly language program. It continues the discussion of parameter passing that appears in earlier chapters in this text. This chapter discusses, in greater detail, several of the different places that a program can pass parameters to a procedure including registers, FPU/MMX register, on the stack, in the code stream, in global variables, and in parameters blocks. While this is not an all-inclusive list, it does cover the more common places where programs pass parameters.

In addition to where, this chapter discusses how programs can pass parameters. Possible ways include pass by value, pass by reference, pass by value/result, pass by result, pass by name, and pass by lazy evaluation. Again, these don't represent all the possible ways one could think of, but it does cover (by far) the most common ways programs pass parameters between procedures.

Another parameter-related issue this chapter discusses is how to pass parameters passed into one procedure as parameters to another procedure. Although HLA's high level calling syntax can take care of the grungy details for you, it's important to know how to pass these parameters manually since there are many instances where you will be forced to write the code that passes these parameters (not to mention, it's a good idea to know how this works, just on general principles).

This chapter also touches on passing a variable number of parameters between procedures and how to return function results from a procedure.

This chapter will not be the last word on parameters. We'll take another look at parameters in the very next chapter when discussing lexical scope.

