Intermediate Procedures Chapter Three

3.1 Chapter Overview

This chapter picks up where the chapter "Introduction to Procedures" in Volume Three leaves off. That chapter presented a high level view of procedures, parameters, and local variables; this chapter takes a look at some of the low-level implementation details. This chapter begins by discussing the CALL instruction and how it affects the stack. Then it discusses activation records and how a program passes parameters to a procedure and how that procedure maintains local (automatic) variables. Next, this chapter presents an in-depth discussion of pass by value and pass by reference parameters. This chapter concludes by discussing procedure variables, procedural parameters, iterators, and the FOREACH..ENDFOR loop.

3.2 Procedures and the CALL Instruction

Most procedural programming languages implement procedures using the call/return mechanism. That is, some code calls a procedure, the procedure does its thing, and then the procedure returns to the caller. The call and return instructions provide the 80x86's *procedure invocation mechanism*. The calling code calls a procedure with the CALL instruction, the procedure returns to the caller with the RET instruction. For example, the following 80x86 instruction calls the HLA Standard Library *stdout.newln* routine:

```
call stdout.newln;
```

stdout.newln prints a carriage return/line feed sequence to the video display and returns control to the instruction immediately following the "call stdout.newln;" instruction.

The HLA language lets you call procedures using a high level language syntax. Specifically, you may call a procedure by simply specifying the procedure's name and (in the case of *stdout.newln*) an empty parameter list. That is, the following is completely equivalent to "call stdout.newln":

```
stdout.newln();
```

The 80x86 CALL instruction does two things. First, it pushes the address of the instruction immediately following the CALL onto the stack; then it transfers control to the address of the specified procedure. The value that CALL pushes onto the stack is known as the *return address*. When the procedure wants to return to the caller and continue execution with the first statement following the CALL instruction, the procedure simply pops the return address off the stack and jumps (indirectly) to that address. Most procedures return to their caller by executing a RET (return) instruction. The RET instruction pops a return address off the stack and transfers control indirectly to the address it pops off the stack.

By default, the HLA compiler automatically places a RET instruction (along with a few other instructions) at the end of each HLA procedure you write. This is why you haven't had to explicitly use the RET instruction up to this point. To disable the default code generation in an HLA procedure, specify the following options when declaring your procedures:

```
procedure ProcName; @noframe; @nodisplay;
begin ProcName;
    .
    .
    .
end ProcName;
```

The @NOFRAME and @NODISPLAY clauses are examples of procedure *options*. HLA procedures support several such options, including RETURNS (See "The HLA RETURNS Option in Procedures" on page 560.), the @NOFRAME, @NODISPLAY, and @NOALIGNSTACKK. You'll see the purpose of @NOALIGNSTACK and a couple of other procedure options a little later in this chapter. These procedure options may appear in any order following the procedure name (and parameters, if any). Note that @NOF-

 © 2001, By Randall Hyde

RAME and @NODISPLAY (as well as @NOALIGNSTACK) may only appear in an actual procedure declaration. You cannot specify these options in an external procedure prototype.

The @NOFRAME option tells HLA that you don't want the compiler to automatically generate entry and exit code for the procedure. This tells HLA not to automatically generate the RET instruction (along with several other instructions).

The @NODISPLAY option tells HLA that it should not allocate storage in procedure's local variable area for a *display*. The display is a mechanism you use to access non-local VAR objects in a procedure. Therefore, a display is only necessary if you nest procedures in your programs. This chapter will not consider the display or nested procedures; for more details on the display and nested procedures see the appropriate chapter in Volume Five. Until then, you can safely specify the @NODISPLAY option on all your procedures. Note that you may specify the @NODISPLAY option independently of the @NOFRAME option. Indeed, for all of the procedures appearing in this text up to this point specifying the @NODISPLAY option makes a lot of sense because none of those procedures have actually used the display. Procedures that have the @NODISPLAY option are a tiny bit faster and a tiny bit shorter than those procedures that do not specify this option.

The following is an example of the minimal procedure:

```
procedure minimal; nodisplay; noframe; noalignstk;
begin minimal;

    ret();

end minimal;
```

If you call this procedure with the CALL instruction, *minimal* will simply pop the return address off the stack and return back to the caller. You should note that a RET instruction is absolutely necessary when you specify the @NOFRAME procedure option[1]. If you fail to put the RET instruction in the procedure, the program will not return to the caller upon encountering the "end minimal;" statement. Instead, the program will fall through to whatever code happens to follow the procedure in memory. The following example program demonstrates this problem:

```
program missingRET;
#include( "stdlib.hhf" );

    // This first procedure has the NOFRAME
    // option but does not have a RET instruction.

    procedure firstProc; @noframe; @nodisplay;
    begin firstProc;

        stdout.put( "Inside firstProc" nl );

    end firstProc;


    // Because the procedure above does not have a
    // RET instruction, it will "fall through" to
    // the following instruction.  Note that there
    // is no call to this procedure anywhere in
    // this program.

    procedure secondProc; @noframe; @nodisplay;
    begin secondProc;
```

1. Strictly speaking, this isn't true. But some mechanism that pops the return address off the stack and jumps to the return address is necessary in the procedure's body.

 © 2001, By Randall Hyde

```
        stdout.put( "Inside secondProc" nl );
        ret();

    end secondProc;


begin missingRET;

    // Call the procedure that doesn't have
    // a RET instruction.

    call firstProc;

end missingRET;
```

Program 3.1 Effect of Missing RET Instruction in a Procedure

Although this behavior might be desirable in certain rare circumstances, it usually represents a defect in most programs. Therefore, if you specify the @NOFRAME option, always remember to explicitly return from the procedure using the RET instruction.

3.3 Procedures and the Stack

Since procedures use the stack to hold the return address, you must exercise caution when pushing and popping data within a procedure. Consider the following simple (and defective) procedure:

```
procedure MessedUp; noframe; nodisplay;
begin MessedUp;

    push( eax );
    ret();

end MessedUp;
```

At the point the program encounters the RET instruction, the 80x86 stack takes the form shown in Figure 3.1:

© 2001, By Randall Hyde

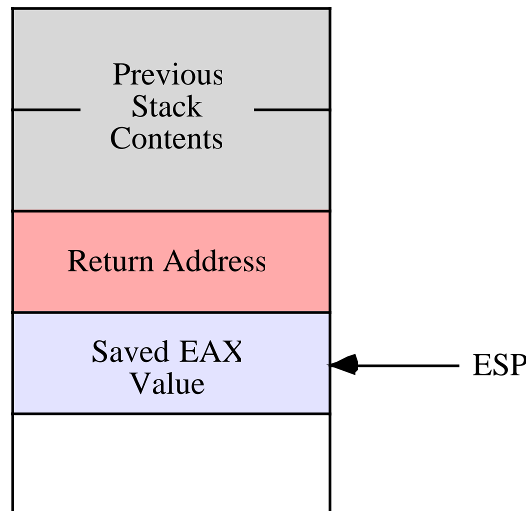

Figure 3.1 Stack Contents Before RET in "MessedUp" Procedure

The RET instruction isn't aware that the value on the top of stack is not a valid address. It simply pops whatever value is on the top of the stack and jumps to that location. In this example, the top of stack contains the saved EAX value. Since it is very unlikely that EAX contains the proper return address (indeed, there is about a one in four billion chance it is correct), this program will probably crash or exhibit some other undefined behavior. Therefore, you must take care when pushing data onto the stack within a procedure that you properly pop that data prior to returning from the procedure.

> **Note:** if you do not specify the @NOFRAME option when writing a procedure, HLA automatically generates code at the beginning of the procedure that pushes some data onto the stack. Therefore, unless you understand exactly what is going on and you've taken care of this data HLA pushes on the stack, you should never execute the bare RET instruction inside a procedure that does not have the @NOFRAME option. Doing so will attempt to return to the location specified by this data (which is not a return address) rather than properly returning to the caller. In procedures that do not have the @NOFRAME option, use the EXIT or EXITIF statements to return from the procedure (See "BEGIN..EXIT..EXITIF..END" on page 740.).

Popping extra data off the stack prior to executing the RET statement can also create havoc in your programs. Consider the following defective procedure:

```
procedure MessedUpToo; noframe; nodisplay;
begin MessedUpToo;

    pop( eax );
    ret();

end MessedUpToo;
```

Upon reaching the RET instruction in this procedure, the 80x86 stack looks something like that shown in Figure 3.2:

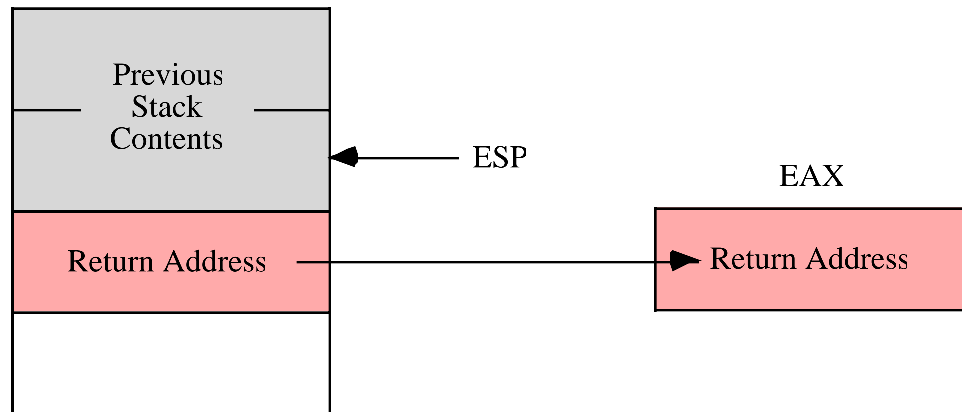

Figure 3.2 Stack Contents Before RET in MessedUpToo

Once again, the RET instruction blindly pops whatever data happens to be on the top of the stack and attempts to return to that address. Unlike the previous example, where it was very unlikely that the top of stack contained a valid return address (since it contained the value in EAX), there is a small possibility that the top of stack in this example actually *does* contain a return address. However, this will not be the proper return address for the *MessedUpToo* procedure; instead, it will be the return address for the procedure that called *MessUpToo*. To understand the effect of this code, consider the following program:

```
program extraPop;
#include( "stdlib.hhf" );


    // Note that the following procedure pops
    // excess data off the stack (in this case,
    // it pops messedUpToo's return address).

    procedure messedUpToo; @noframe; @nodisplay;
    begin messedUpToo;

        stdout.put( "Entered messedUpToo" nl );
        pop( eax );
        ret();

    end messedUpToo;



    procedure callsMU2; @noframe; @nodisplay;
    begin callsMU2;

        stdout.put( "calling messedUpToo" nl );
        messedUpToo();

        // Because messedUpToo pops extra data
        // off the stack, the following code
        // never executes (since the data popped
        // off the stack is the return address that
        // points at the following code.
```

```
        stdout.put( "Returned from messedUpToo" nl );
        ret();

    end callsMU2;


begin extraPop;

    stdout.put( "Calling callsMU2" nl );
    callsMU2();
    stdout.put( "Returned from callsMU2" nl );

end extraPop;
```

Program 3.2 Effect of Popping Too Much Data Off the Stack

Since a valid return address is sitting on the top of the stack, you might think that this program will actually work (properly). However, note that when returning from the *MessedUpToo* procedure, this code returns directly to the main program rather than to the proper return address in the *EndSkipped* procedure. Therefore, all code in the *callsMU2* procedure that follows the call to *MessedUpToo* does not execute. When reading the source code, it may be very difficult to figure out why those statements are not executing since they immediately follow the call to the *MessUpToo* procedure. It isn't clear, unless you look very closely, that the program is popping an extra return address off the stack and, therefore, doesn't return back to *callsMU2* but, rather, returns directly to whomever calls *callsMU2*. Of course, in this example it's fairly easy to see what is going on (because this example is a demonstration of this problem). In real programs, however, determining that a procedure has accidentally popped too much data off the stack can be much more difficult. Therefore, you should always be careful about pushing and popping data in a procedure. You should always verify that there is a one-to-one relationship between the pushes in your procedures and the corresponding pops.

3.4 Activation Records

Whenever you call a procedure there is certain information the program associates with that procedure call. The return address is a good example of some information the program maintains for a specific procedure call. Parameters and automatic local variables (i.e., those you declare in the VAR section) are additional examples of information the program maintains for each procedure call. *Activation record* is the term we'll use to describe the information the program associates with a specific call to a procedure[2].

Activation record is an appropriate name for this data structure. The program creates an activation record when calling (activating) a procedure and the data in the structure is organized in a manner identical to records (see "Records" on page 483). Perhaps the only thing unusual about an activation record (when comparing it to a standard record) is that the base address of the record is in the middle of the data structure, so you must access fields of the record at positive and negative offsets.

Construction of an activation record begins in the code that calls a procedure. The caller pushes the parameter data (if any) onto the stack. Then the execution of the CALL instruction pushes the return address onto the stack. At this point, construction of the activation record continues withinin the procedure itself. The procedure pushes registers and other important state information and then makes room in the activation record for local variables. The procedure must also update the EBP register so that it points at the base address of the activation record.

2. Stack frame is another term many people use to describe the activation record.

 © 2001, By Randall Hyde

To see what a typical activation record looks like, consider the following HLA procedure declaration:

```
procedure ARDemo( i:uns32; j:int32; k:dword ); nodisplay;
var
    a:int32;
    r:real32;
    c:char;
    b:boolean;
    w:word;
begin ARDemo;
    .
    .
    .
end ARDemo;
```

Whenever an HLA program calls this *ARDemo* procedure, it begins by pushing the data for the parameters onto the stack. The calling code will push the parameters onto the stack in the order they appear in the parameter list, from left to right. Therefore, the calling code first pushes the value for the *i* parameter, then it pushes the value for the *j* parameter, and it finally pushes the data for the *k* parameter. After pushing the parameters, the program calls the *ARDemo* procedure. Immediately upon entry into the ARDemo procedure, the stack contains these four items arranged as shown in Figure 3.3

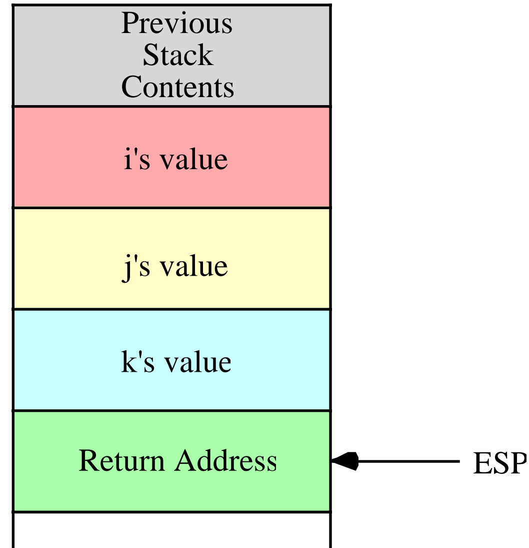

Figure 3.3 Stack Organization Immediately Upon Entry into ARDemo

The first few instructions in *ARDemo* (note that it does not have the @NOFRAME option) will push the current value of EBP onto the stack and then copy the value of ESP into EBP. Next, the code drops the stack pointer down in memory to make room for the local variables. This produces the stack organization shown in Figure 3.4

 © 2001, By Randall Hyde

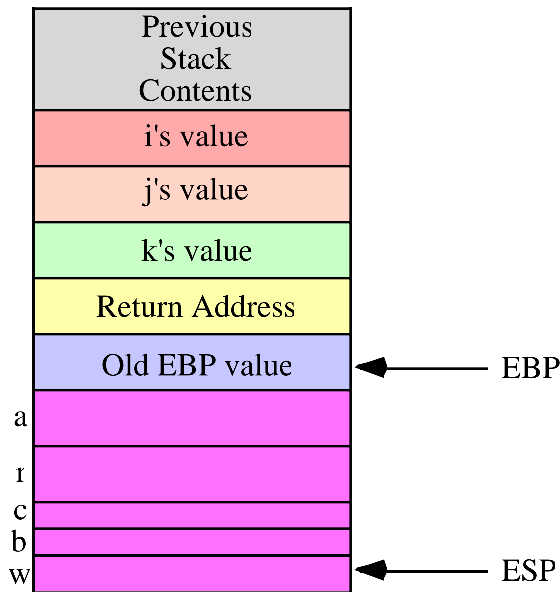

Figure 3.4 Activation Record for ARDemo

To access objects in the activation record you must use offsets from the EBP register to the desired object. The two items of immediate interest to you are the parameters and the local variables. You can access the parameters at positive offsets from the EBP register, you can access the local variables at negative offsets from the EBP register as Figure 3.5 shows:

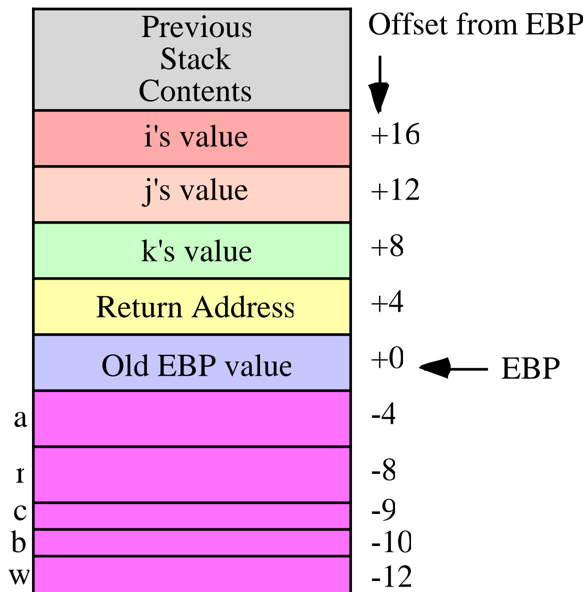

Figure 3.5 Offsets of Objects in the ARDemo Activation Record

Intel specifically reserves the EBP (extended base pointer) for use as a pointer to the base of the activation record. This is why you should never use the EBP register for general calculations. If you arbitrarily

 © 2001, By Randall Hyde

change the value in the EBP register you will lose access to the current procedure's parameters and local variables.

3.5 The Standard Entry Sequence

The caller of a procedure is responsible for pushing the parameters onto the stack. Of course, the CALL instruction pushes the return address onto the stack. It is the procedure's responsibility to construct the rest of the activation record. This is typically accomplished by the following "standard entry sequence" code:

```
push( ebp );         // Save a copy of the old EBP value
mov( esp, ebp );     // Get ptr to base of activation record into EBP
sub( NumVars, esp ); // Allocate storage for local variables.
```

If the procedure doesn't have any local variables, the third instruction above, "sub(NumVars, esp);" isn't needed. *NumVars* represents the number of bytes of local variables needed by the procedure. This is a constant that should be an even multiple of four (so the ESP register remains aligned on a double word boundary). If the number of bytes of local variables in the procedure is not an even multiple of four, you should round the value up to the next higher multiple of four before subtracting this constant from ESP. Doing so will slightly increase the amount of storage the procedure uses for local variables but will not otherwise affect the operation of the procedure.

> **Warning**: if the *NumVars* constant is not an even multiple of four, subtracting this value from ESP (which, presumably, contains a dword-aligned pointer) will virtually guarantee that all future stack accesses are misaligned since the program almost always pushes and pops dword values. This will have a very negative performance impact on the program. Worse still, many OS API calls will fail if the stack is not dword-aligned upon entry into the operating system. Therefore, you must always ensure that your local variable allocation value is an even multiple of four.

Because of the problems with a misaligned stack, by default HLA will also emit a fourth instruction as part of the standard entry sequence. The HLA compiler actually emits the following standard entry sequence for the ARDemo procedure defined earlier:

```
push( ebp );
mov( esp, ebp );
sub( 12, esp );          // Make room for ARDemo's local variables.
and( $FFFF_FFFC, esp );  // Force dword stack alignment.
```

The AND instruction at the end of this sequence forces the stack to be aligned on a four-byte boundary (it reduces the value in the stack pointer by one, two, or three if the value in ESP is not an even multiple of four). Although the *ARDemo* entry code correctly subtracts 12 from ESP for the local variables (12 is both an even multiple of four and the number of bytes of local variables), this only leaves ESP double word aligned if it was double word aligned immediately upon entry into the procedure. Had the caller messed with the stack and left ESP containing a value that was not an even multiple of four, subtracting 12 from ESP would leave ESP containing an unaligned value. The AND instruction in the sequence above, however, guarantees that ESP is dword aligned regardless of ESP's value upon entry into the procedure. The few bytes and CPU cycles needed to execute this instruction pay off handsomely if ESP is not double word aligned.

Although it is always safe to execute the AND instruction in the standard entry sequence, it might not be necessary. If you always ensure that ESP contains a double word aligned value, the AND instruction in the standard entry sequence above is unnecessary. Therefore, if you've specified the @NOFRAME procedure option, you don't have to include that instruction as part of the entry sequence.

If you haven't specified the @NOFRAME option (i.e., you're letting HLA emit the instructions to construct the standard entry sequence for you), you can still tell HLA not to emit the extra AND instruction if you're sure the stack will be dword aligned whenever someone calls the procedure. To do this, use the @NOALIGNSTACK procedure option, e.g.,

```
procedure NASDemo( i:uns32; j:int32; k:dword ); @noalignstack;
```

 © 2001, By Randall Hyde

```
var
    LocalVar:int32;
begin NASDemo;
    .
    .
    .
end NASDemo;
```

HLA emits the following entry sequence for the procedure above:

```
push( ebp );
mov( esp, ebp );
sub( 4, esp );
```

3.6 The Standard Exit Sequence

Before a procedure returns to its caller, it needs to clean up the activation record. Although it is possible to share the clean-up duties between the procedure and the procedure's caller, Intel has included some features in the instruction set that allows the procedure to efficiently handle all the clean up chores itself. Standard HLA procedures and procedure calls, therefore, assume that it is the procedure's responsibility to clean up the activation record (including the parameters) when the procedure returns to its caller.

If a procedure does not have any parameters, the calling sequence is very simple. It requires only three instructions:

```
mov( ebp, esp );    // Deallocate locals and clean up stack.
pop( ebp );         // Restore pointer to caller's activation record.
ret();              // Return to the caller.
```

If the procedure has some parameters, then a slight modification to the standard exit sequence is necessary in order to remove the parameter data from the stack. Procedures with parameters use the following standard exit sequence:

```
mov( ebp, esp );    // Deallocate locals and clean up stack.
pop( ebp );         // Restore pointer to caller's activation record.
ret( ParmBytes );   // Return to the caller and pop the parameters.
```

The *ParmBytes* operand of the RET instruction is a constant that specifies the number of *bytes* of parameter data to remove from the stack after the return instruction pops the return address. For example, the ARDemo example code in the previous sections has three double word parameters. Therefore, the standard exit sequence would take the following form:

```
mov( ebp, esp );
pop( ebp );
ret( 12 );
```

If you've declared your parameters using HLA syntax (i.e., a parameter list follows the procedure declaration), then HLA automatically creates a local constant in the procedure, *_parms_*, that is equal to the number of bytes of parameters in that procedure. Therefore, rather than worrying about having to count the number of parameter bytes yourself, you can use the following standard exit sequence for any procedure that has parameters:

```
mov( ebp, esp );
pop( ebp );
ret( _parms_ );
```

Note that if you do not specify a byte constant operand to the RET instruction, the 80x86 will not pop the parameters off the stack upon return. Those parameters will still be sitting on the stack when you execute the first instruction following the CALL to the procedure. Similarly, if you specify a value that is too

 © 2001, By Randall Hyde

small, some of the parameters will be left on the stack upon return from the procedure. If the RET operand you specify is too large, the RET instruction will actually pop some of the caller's data off the stack, usually with disastrous consequences.

Note that if you wish to return early from a procedure that doesn't have the @NOFRAME option, and you don't particularly want to use the EXIT or EXITIF statement, you must execute the standard exit sequence to return to the caller. A simple RET instruction is insufficient since local variables and the old EBP value are probably sitting on the top of the stack.

3.7 HLA Local Variables

Your program accesses local variables in a procedure by using negative offsets from the activation record base address (EBP). For example, consider the following HLA procedure (which admittedly, doesn't do much other than demonstrate the use of local variables):

```
procedure LocalVars; nodisplay;
var
    a:int32;
    b:int32;
begin LocalVars;

    mov( 0, a );
    mov( a, eax );
    mov( eax, b );

end LocalVars;
```

The activation record for LocalVars looks like

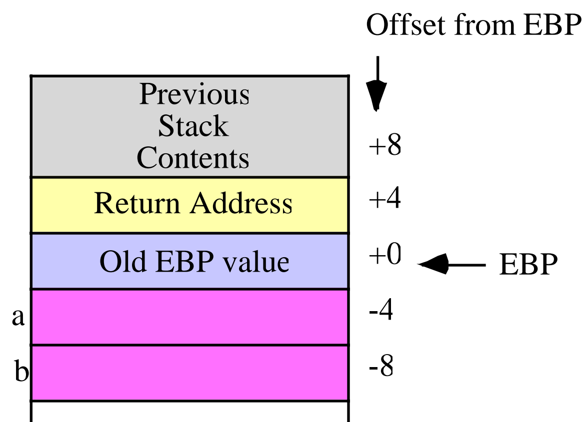

Figure 3.6 Activation Record for LocalVars Procedure

The HLA compiler emits code that is roughly equivalent to the following for the body of this procedure[3]:

```
mov( 0, (type dword [ebp-4]));
mov( [ebp-4], eax );
mov( eax, [ebp-8] );
```

3. Ignoring the code associated with the standard entry and exit sequences.

 © 2001, By Randall Hyde

You could actually type these statements into the procedure yourself and they would work. Of course, using memory references like "[ebp-4]" and "[ebp-8]" rather than *a* or *b* makes your programs very difficult to read and understand. Therefore, you should always declare and use HLA symbolic names rather than offsets from EBP.

The standard entry sequence for this *LocalVars* procedure will be[4]

```
push( ebp );
mov( esp, ebp );
sub( 8, esp );
```

This code subtracts eight from the stack pointer because there are eight bytes of local variables (two dword objects) in this procedure. Unfortunately, as the number of local variables increases, especially if those variables have different types, computing the number of bytes of local variables becomes rather tedious. Fortunately, for those who wish to write the standard entry sequence themselves, HLA automatically computes this value for you and creates a constant, *_vars_*, that specifies the number of bytes of local variables for you[5]. Therefore, if you intend to write the standard entry sequence yourself, you should use the *_vars_* constant in the SUB instruction when allocating storage for the local variables:

```
push( ebp );
mov( esp, ebp );
sub( _vars_, esp );
```

Now that you've seen how assembly language (and, indeed, most languages) allocate and deallocate storage for local variables, it's easy to understand why automatic (local VAR) variables do not maintain their values between two calls to the same procedure. Since the memory associated with these automatic variables is on the stack, when a procedure returns to its caller the caller can push other data onto the stack obliterating the values of the local variable values previously held on the stack. Furthermore, intervening calls to other procedures (with their own local variables) may wipe out the values on the stack. Also, upon reentry into a procedure, the procedure's local variables may correspond to different physical memory locations, hence the values of the local variables would not be in their proper locations.

One big advantage to automatic storage is that it efficiently shares a fixed pool of memory among several procedures. For example, if you call three procedures in a row,

```
ProcA();
ProcB();
ProcC();
```

The first procedure (*ProcA* in the code above) allocates its local variables on the stack. Upon return, *ProcA* deallocates that stack storage. Upon entry into *ProcB*, the program allocates storage for *ProcB's* local variables *using the same memory locations just freed by ProcA*. Likewise, when *ProcB* returns and the program calls *ProcC*, *ProcC* uses the same stack space for its local variables that *ProcB* recently freed up. This memory reuse makes efficient use of the system resources and is probably the greatest advantage to using automatic (VAR) variables.

3.8 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:

4. This code assumes that ESP is dword aligned upon entry so the "AND($FFFF_FFFC, ESP);" instruction is unnecessary.
5. HLA even rounds this constant up to the next even multiple of four so you don't have to worry about stack alignment.

 © 2001, By Randall Hyde

- *where* is the data coming from?
- *what* mechanism do you use to pass and return data?
- *how* much data are you passing?

In this chapter we will take another look at the two most common parameter passing mechanisms: pass by value and pass by reference. We will also discuss three popular places to pass parameters: in the registers, on the stack, and in the code stream. The amount of parameter data has a direct bearing on where and how to pass it. The following sections take up these issues.

3.8.1 Pass by Value

A parameter passed by value is just that – the caller passes a value to the procedure. Pass by value parameters are input only parameters. That is, you can pass them to a procedure but the procedure cannot return values through them. In high level languages the idea of a pass by value parameter being an input only parameter makes a lot of sense. Given the procedure call:

```
CallProc(I);
```

If you pass *I* by value, *CallProc* does not change the value of *I*, regardless of what happens to the parameter inside *CallProc*.

Since you must pass a copy of the data to the procedure, you should only use this method for passing small objects like bytes, words, and double words. Passing arrays and strings by value is very inefficient (since you must create and pass a copy of the structure to the procedure).

3.8.2 Pass by Reference

To pass a parameter by reference you must pass the address of a variable rather than its value. In other words, you must pass a pointer to the data. The procedure must dereference this pointer to access the data. Passing parameters by reference is useful when you must modify the actual parameter or when you pass large data structures between procedures.

Passing parameters by reference can produce some peculiar results. The following Pascal procedure provides an example of one problem you might encounter:

```
program main(input,output);
var m:integer;

        (*
        ** Note: this procedure passes i and j by reference.
        *)

        procedure bletch(var i,j:integer);
        begin

            i := i+2;
            j := j-i;
            writeln(i,' ',j);

        end;

         .
         .
         .

    begin {main}

        m := 5;
        bletch(m,m);

    end.
```

This particular code sequence will print "00" regardless of *m's* value. This is because the parameters *i* and *j* are pointers to the actual data and they both point at the same object (that is, they are aliases). Therefore, the statement "j:=j-i;" always produces zero since *i* and *j* refer to the same variable.

Pass by reference is usually less efficient than pass by value. You must dereference all pass by reference parameters on each access; this is slower than simply using a value. However, when passing a large data structure, pass by reference is faster because you do not have to copy a large data structure before calling the procedure.

3.8.3 Passing Parameters in Registers

Having touched on how to pass parameters to a procedure, the next thing to discuss is *where* to pass parameters. Where you pass parameters depends 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 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 not a hard and fast rule. If you find it more convenient to pass 16 bit values in the SI or BX register, 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 double words, perhaps you should pass your values elsewhere.

As an example, consider the following "strfill(str,c);" that copies the character *c* (passed by value in AL) to each character position in *s* (passed by reference in EDI) up to a zero terminating byte:

```
// strfill-  Overwrites the data in a string with a character.
//
//    EDI-   pointer to zero terminated string (e.g., an HLA string)
//    AL-    character to store into the string.

procedure strfill; nodisplay;
begin strfill;

    push( edi );  // Preserve this because it will be  modified.
    while( (type char [edi] <> #0 ) do

        mov( al, [edi] );
        inc( edi );

    endwhile;
    pop( edi );

end strfill;
```

 © 2001, By Randall Hyde

To call the *strfill* procedure you would load the address of the string data into EDI and the character value into AL prior to the call. The following code fragment demonstrates a typical call to *strfill*:

```
mov( s, edi );  // Get ptr to string data into edi (assumes s:string).
mov( ' ', al );
strfill();
```

Don't forget that HLA string variables are pointers. This example assumes that *s* is a HLA string variable and, therefore, contains a pointer to a zero-terminated string. Therefore, the "mov(s, edi);" instruction loads the address of the zero terminated string into the EDI register (hence this code passes the address of the string data to *strfill*, that is, it passes the string by reference).

One way to pass parameters in the registers is to simply load the registers with the appropriate values prior to a call and then reference the values in those registers within the procedure. This is the traditional mechanism for passing parameters in registers in an assembly language program. HLA, being somewhat more high level than traditional assembly language, provides a formal parameter declaration syntax that lets you tell HLA you're passing certain parameters in the general purpose registers. This declaration syntax is the following:

```
parmName: parmType in reg
```

Where parmName is the parameter's name, *parmType* is the type of the object, and *reg* is one of the 80x86's general purpose eight, sixteen, or thirty-two bit registers. The size of the parameter's type must be equal to the size of the register or HLA will generate an error. Here is a concrete example:

```
procedure HasRegParms( count: uns32 in ecx; charVal:char in al );
```

One nice feature to this syntax is that you can call a procedure that has register parameters exactly like any other procedure in HLA using the high level syntax, e.g.,

```
HasRegParms( ecx, bl );
```

If you specify the same register as an actual parameter that you've declared for the formal parameter, HLA does not emit any extra code; it assumes that the parameter is already in the appropriate register. For example, in the call above the first actual parameter is the value in ECX; since the procedure's declaration specifies that that first parameter is in ECX HLA will not emit any code. On the other hand, the second actual parameter is in BL while the procedure will expect this parameter value in AL. Therefore, HLA will emit a "mov(bl, al);" instruction prior to calling the procedure so that the value is in the proper register upon entry to the procedure.

You can also pass parameters by reference in a register. Consider the following declaration:

```
procedure HasRefRegParm( var myPtr:uns32 in edi );
```

A call to this procedure always requires some memory operand as the actual parameter. HLA will emit the code to load the address of that memory object into the parameter's register (EDI in this case). Note that when passing reference parameters, the register must be a 32-bit general purpose register since addresses are 32-bits long. Here's an example of a call to *HasRefRegParm*:

```
HasRefRegParm( x );
```

HLA will emit either a "mov(&x, edi);" or "lea(edi, x);" instruction to load the address of *x* into the EDI registers prior to the CALL instruction[6].

If you pass an anonymous memory object (e.g., "[edi]" or "[ecx]") as a parameter to HasRefRegParm, HLA will not emit any code if the memory reference uses the same register that you declare for the parameter (i.e., "[edi]"). It will use a simple MOV instruction to copy the actual address into EDI if you specify an indirect addressing mode using a register other than EDI (e.g., "[ecx]"). It will use an LEA instruction to compute the effective address of the anonymous memory operand if you use a more complex addressing mode like "[edi+ecx*4+2]".

6. The choice of instructions is dictated by whether *x* is a static variable (MOV for static objects, LEA for other objects).

 © 2001, By Randall Hyde

Within the procedure's code, HLA creates text equates for these register parameters that map their names to the appropriate register. In the *HasRegParms* example, any time you reference the *count* parameter, HLA substitutes "ecx" for *count*. Likewise, HLA substitutes "al" for *charVal* throughout the procedure's body. Since these names are aliases for the registers, you should take care to always remember that you cannot use ECX and AL independently of these parameters. It would be a good idea to place a comment next to each use of these parameters to remind the reader that *count* is equivalent to ECX and *charVal* is equivalent to AL.

3.8.4 Passing Parameters in the Code Stream

Another place where you can pass parameters is in the code stream immediately after the CALL instruction. Consider the following *print* routine that prints a literal string constant to the standard output device:

```
call print;
byte "This parameter is in the code stream.",0;
```

Normally, a subroutine returns control to the first instruction immediately following the CALL instruction. Were that to happen here, the 80x86 would attempt to interpret the ASCII codes for "This..." as an instruction. This would produce undesirable results. Fortunately, you can skip over this string when returning from the subroutine.

So how do you gain access to these parameters? Easy. The return address on the stack points at them. Consider the following implementation of print:

```
program printDemo;
#include( "stdlib.hhf" );

    // print-
    //
    //  This procedure writes the literal string
    //  immediately following the call to the
    //  standard output device.  The literal string
    //  must be a sequence of characters ending with
    //  a zero byte (i.e., a C string, not an HLA
    //  string).

    procedure print; @noframe; @nodisplay;
    const

        // RtnAdrs is the offset of this procedure's
        // return address in the activation record.

        RtnAdrs:text := "(type dword [ebp+4])";

    begin print;

        // Build the activation record (note the
        // "@noframe" option above).

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

        // Preserve the registers this function uses.

        push( eax );
        push( ebx );

        // Copy the return address into the EBX
```

```
        // register.  Since the return address points
        // at the start of the string to print, this
        // instruction loads EBX with the address of
        // the string to print.

        mov( RtnAdrs, ebx );

        // Until we encounter a zero byte, print the
        // characters in the string.

        forever

            mov( [ebx], al );   // Get the next character.
            breakif( !al );     // Quit if it's zero.
            stdout.putc( al );  // Print it.
            inc( ebx );         // Move on to the next char.

        endfor;

        // Skip past the zero byte and store the resulting
        // address over the top of the return address so
        // we'll return to the location that is one byte
        // beyond the zero terminating byte of the string.

        inc( ebx );
        mov( ebx, RtnAdrs );

        // Restore EAX and EBX.

        pop( ebx );
        pop( eax );

        // Clean up the activation record and return.

        pop( ebp );
        ret();

    end print;


begin printDemo;

    // Simple test of the print procedure.

    call print;
    byte "Hello World!", 13, 10, 0 ;

end printDemo;
```

Program 3.3 Print Procedure Implementation (Using Code Stream Parameters)

Besides showing how to pass parameters in the code stream, the *print* routine also exhibits another concept: *variable length parameters*. The string following the CALL can be any practical length. The zero terminating byte marks the end of the parameter list. There are two easy ways to handle variable length parameters. Either use some special terminating value (like zero) or you can pass a special length value that tells the subroutine how many parameters you are passing. Both methods have their advantages and disadvantages. Using a special value to terminate a parameter list requires that you choose a value that never

appears in the list. For example, *print* uses zero as the terminating value, so it cannot print the NUL character (whose ASCII code is zero). Sometimes this isn't a limitation. Specifying a special length parameter is another mechanism you can use to pass a variable length parameter list. While this doesn't require any special codes or limit the range of possible values that can be passed to a subroutine, setting up the length parameter and maintaining the resulting code can be a real nightmare[7].

Despite the convenience afforded by passing parameters in the code stream, there are some disadvantages to passing parameters there. First, if you fail to provide the exact number of parameters the procedure requires, the subroutine will get very confused. Consider the *print* example. It prints a string of characters up to a zero terminating byte and then returns control to the first instruction following the zero terminating byte. If you leave off the zero terminating byte, the *print* routine happily prints the following opcode bytes as ASCII characters until it finds a zero byte. Since zero bytes often appear in the middle of an instruction, the *print* routine might return control into the middle of some other instruction. This will probably crash the machine. Inserting an extra zero, which occurs more often than you might think, is another problem programmers have with the *print* routine. In such a case, the *print* routine would return upon encountering the first zero byte and attempt to execute the following ASCII characters as machine code. Once again, this usually crashes the machine. These are the some of the reasons why the HLA *stdout.put* code does *not* pass its parameters in the code stream. Problems notwithstanding, however, the code stream is an efficient place to pass parameters whose values do not change.

3.8.5 Passing Parameters on the Stack

Most high level languages use the stack to pass parameters because this method is fairly efficient. By default, HLA also passes parameters on the stack. Although passing parameters on the stack is slightly less efficient than passing those parameters in registers, the register set is very limited and you can only pass a few value or reference parameters through registers. The stack, on the other hand, allows you to pass a large amount of parameter data without any difficulty. This is the principal reason that most programs pass their parameters on the stack.

HLA passes parameters you specify in a high-level language form on the stack. For example, suppose you define *strfill* from the previous section as follows:

```
procedure strfill( s:string; chr:char );
```

Calls of the form "strfill(s, ' ');" will pass the value of *s* (which is an address) and a space character on the 80x86 stack. When you specify a call to *strfill* in this manner, HLA automatically pushes the parameters for you, so you don't have to push them onto the stack yourself. Of course, if you choose to do so, HLA will let you manually push the parameters onto the stack prior to the call.

To manually 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 call:

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

HLA pushes parameters onto the stack in the order that they appear in the parameter list[8]. Therefore, the 80x86 code HLA emits for this subroutine call (assuming you're passing the parameters by value) is

```
push( i );
push( j );
push( k );
call CallProc;
```

Upon entry into CallProc, the 80x86's stack looks like that shown in Figure 3.7:

7. Especially if the parameter list changes frequently.

8. Assuming, of course, that you don't instruct HLA otherwise. It is possible to tell HLA to reverse the order of the parameters on the stack. See the chapter on "Mixed Language Programming" for more details.

 © 2001, By Randall Hyde

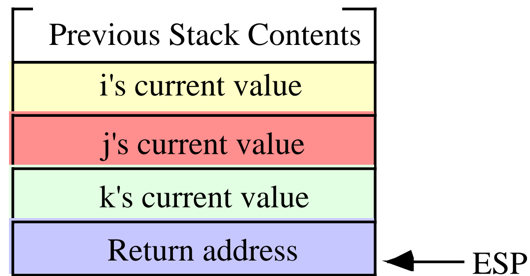

Figure 3.7 Stack Layout Upon Entry into CallProc

You could gain access to the parameters passed on the stack by removing the data from the stack as the following code fragment demonstrates:

```
// Note: to extract parameters off the stack by popping it is very important
// to specify both the @nodisplay and @noframe procedure options.

static
    RtnAdrs: dword;
    p1Parm: dword;
    p2Parm: dword;
    p3Parm: dword;

procedure CallProc( p1:dword; p2:dword; p3:dword ); @nodisplay; @noframe;
begin CallProc;

    pop( RtnAdrs );
    pop( p3Parm );
    pop( p2Parm );
    pop( p1Parm );
    push( RtnAdrs );
        .
        .
        .
    ret();

end CallProc;
```

As you can see from this code, it first pops the return address off the stack and into the *RtnAdrs* variable; then it pops (in reverse order) the values of the *p1*, *p2*, and *p3* parameters; finally, it pushes the return address back onto the stack (so the RET instruction will operate properly). Within the *CallProc* procedure, you may access the *p1Parm*, *p2Parm*, and *p3Parm* variables to use the *p1*, *p2*, and *p3* parameter values.

There is, however, a better way to access procedure parameters. If your procedure includes the standard entry and exit sequences (see “The Standard Entry Sequence” on page 813 and “The Standard Exit Sequence” on page 814), then you may directly access the parameter values in the activation record by indexing off the EBP register. Consider the layout of the activation record for *CallProc* that uses the following declaration:

```
procedure CallProc( p1:dword; p2:dword; p3:dword ); @nodisplay; @noframe;
begin CallProc;

    push( ebp );     // This is the standard entry sequence.
    mov( esp, ebp ); // Get base address of A.R. into EBP.
```

```
.
.
.
```

Take a look at the stack immediately after the execution of "mov(esp, ebp);" in *CallProc*. Assuming you've pushed three double word parameters onto the stack, it should look something like shown in Figure 3.8:

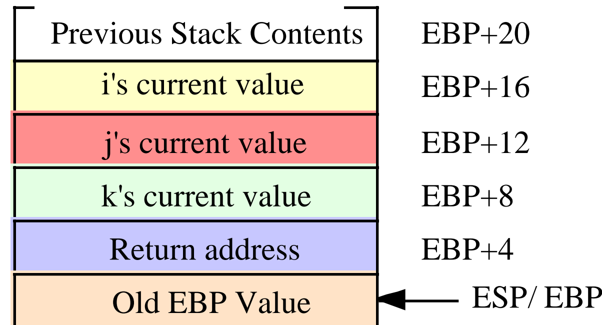

Figure 3.8 Activation Record for CallProc After Standard Entry Sequence Execution

.Now you can access the parameters by indexing off the EBP register:

```
mov( [ebp+16], eax );   // Accesses the first parameter.
mov( [ebp+12], ebx );   // Accesses the second parameter.
mov( [ebp+8], ecx );    // Accesses the third parameter.
```

Of course, like local variables, you'd never really access the parameters in this way. You can use the formal parameter names (*p1*, *p2*, and *p3*) and HLA will substitute a suitable "[ebp+*displacement*]" memory address. Even though you shouldn't actually access parameters using address expressions like "[ebp+12]" it's important to understand their relationship to the parameters in your procedures.

Other items that often appear in the activation record are register values your procedure preserves. The most rational place to preserve registers in a procedure is in the code immediately following the standard entry sequence. In a standard HLA procedure (one where you do not specify the NOFRAME option), this simply means that the code that preserves the registers should appear first in the procedure's body. Likewise, the code to restore those register values should appear immediately before the END clause for the procedure[9].

3.8.5.1 Accessing Value Parameters on the Stack

Accessing parameters passed by value is no different than accessing a local VAR object. As long as you've declared the parameter in a formal parameter list and the procedure executes the standard entry sequence upon entry into the program, all you need do is specify the parameter's name to reference the value of that parameter. The following is an example program whose procedure accesses a parameter the main program passes to it by value:

```
program AccessingValueParameters;
```

9. Note that if you use the EXIT statement to exit a procedure, you must duplicate the code to pop the register values and place this code immediately before the EXIT clause. This is a good example of a maintenance nightmare and is also a good reason why you should only have one exit point in your program.

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

    procedure ValueParm( theParameter: uns32 ); @nodisplay;
    begin ValueParm;

        mov( theParameter, eax );
        add( 2, eax );
        stdout.put
        (
            "theParameter + 2 = ",
            (type uns32 eax),
            nl
        );

    end ValueParm;


begin AccessingValueParameters;

    ValueParm( 10 );
    ValueParm( 135 );

end AccessingValueParameters;
```

Program 3.4 Demonstration of Value Parameters

Although you may access the value of *theParameter* using the anonymous address "[EBP+8]" within your code, there is absolutely no good reason for doing so. If you declare the parameter list using the HLA high level language syntax, you can access the value parameter by specifying its name within the procedure.

3.8.5.2 Passing Value Parameters on the Stack

As Program 3.4 demonstrates, passing a value parameter to a procedure is very easy. Just specify the value in the actual parameter list as you would for a high level language call. Actually, the situation is a little more complicated than this. Passing value parameters is easy if you're passing constant, register, or variable values. It gets a little more complex if you need to pass the result of some expression. This section deals with the different ways you can pass a parameter by value to a procedure.

Of course, you do not have to use the HLA high level language syntax to pass value parameters to a procedure. You can push these values on the stack yourself. Since there are many times it is more convenient or more efficient to manually pass the parameters, describing how to do this is a good place to start.

As noted earlier in this chapter, when passing parameters on the stack you push the objects in the order they appear in the formal parameter list (from left to right). When passing parameters by value, you should push the values of the actual parameters onto the stack. The following program demonstrates how to do this:

```
program ManuallyPassingValueParameters;
#include( "stdlib.hhf" )

    procedure ThreeValueParms( p1:uns32; p2:uns32; p3:uns32 ); @nodisplay;
    begin ThreeValueParms;

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


© 2001, By Randall Hyde

```
        add( p3, eax );
        stdout.put
        (
            "p1 + p2 + p3 = ",
            (type uns32 eax),
            nl
        );

    end ThreeValueParms;


static
    SecondParmValue:uns32 := 25;

begin ManuallyPassingValueParameters;

    pushd( 10 );                // Value associated with p1.
    pushd( SecondParmValue);    // Value associated with p2.
    pushd( 15 );                // Value associated with p3.
    call ThreeValueParms;

end ManuallyPassingValueParameters;
```

Program 3.5 Manually Passing Parameters on the Stack

Note that if you manually push the parameters onto the stack as this example does, you must use the CALL instruction to call the procedure. If you attempt to use a procedure invocation of the form "ThreeValueParms();" then HLA will complain about a mismatched parameter list. HLA won't realize that you've manually pushed the parameters (as far as HLA is concerned, those pushes appear to preserve some other data).

Generally, there is little reason to manually push a parameter onto the stack if the actual parameter is a constant, a register value, or a variable. HLA's high level syntax handles most such parameters for you. There are several instances, however, where HLA's high level syntax won't work. The first such example is passing the result of an arithmetic expression as a value parameter. Since arithmetic expressions don't exist in HLA, you will have to manually compute the result of the expression and pass that value yourself. There are two possible ways to do this: calculate the result of the expression and manually push that result onto the stack, or compute the result of the expression into a register and pass the register as a parameter to the procedure. Program 3.6 demonstrates these two mechanisms.

```
program PassingExpressions;
#include( "stdlib.hhf" )

    procedure ExprParm( exprValue:uns32 ); @nodisplay;
    begin ExprParm;

        stdout.put( "exprValue = ", exprValue, nl );

    end ExprParm;


static
    Operand1: uns32 := 5;
    Operand2: uns32 := 20;

begin PassingExpressions;
```

 © 2001, By Randall Hyde

```
        // ExprParm( Operand1 + Operand2 );
        //
        //  Method one: Compute the sum and manually
        //  push the sum onto the stack.

        mov( Operand1, eax );
        add( Operand2, eax );
        push( eax );
        call ExprParm;

        //  Method two: Compute the sum in a register and
        //  pass the register using the HLA high level
        //  language syntax.

        mov( Operand1, eax );
        add( Operand2, eax );
        ExprParm( eax );

    end PassingExpressions;
```

Program 3.6 Passing the Result of Some Arithmetic Expression as a Parameter

The examples up to this point in this section have made an important assumption: that the parameter you are passing is a double word value. The calling sequence changes somewhat if you're passing parameters that are not four-byte objects. Because HLA can generate relatively inefficient code when passing objects that are not four-bytes long, manually passing such objects is a good idea if you want to have the fastest possible code.

HLA requires that all value parameters be an even multiple of four bytes long[10]. If you pass an object that is less than four bytes long, HLA requires that you *pad* the parameter data with extra bytes so that you always pass an object that is at least four bytes in length. For parameters that are larger than four bytes, you must ensure that you pass an even multiple of four bytes as the parameter value, adding extra bytes at the high-order end of the object to pad it, as necessary.

Consider the following procedure prototype:

```
procedure OneByteParm( b:byte );
```

The activation record for this procedure looks like the following:

10. This only applies if you use the HLA high level language syntax to declare and access parameters in your procedures. Of course, if you manually push the parameters yourself and you access the parameters inside the procedure using an addressing mode like "[ebp+8]" then you can pass any sized object you choose. Of course, keep in mind that most operating systems expect the stack to be dword-aligned, so parameters you push should be a multiple of four bytes long.

 © 2001, By Randall Hyde

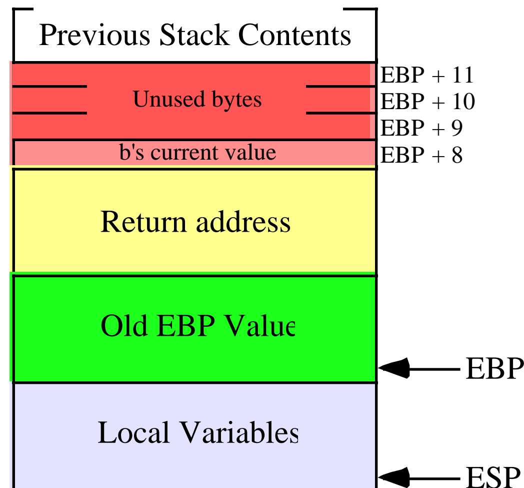

Figure 3.9 OneByteParm Activation Record

As you can see, there are four bytes on the stack associated with the *b* parameter, but only one of the four bytes contains valid data (the L.O. byte). The remaining three bytes are just padding and the procedure should ignore these bytes. In particular, you should never assume that these extra bytes contain zeros or some other consistent value. Depending on the type of parameter you pass, HLA's automatic code generation may or may not push zero bytes as the extra data on the stack.

When passing a byte parameter to a procedure, HLA will automatically emit code that pushes four bytes on the stack. Because HLA's parameter passing mechanism guarantees not to disturb any register or other values, HLA often generates more code than is actually needed to pass a byte parameter. For example, if you decide to pass the AL register as the byte parameter, HLA will emit code that pushes the EAX register onto the stack. This single push instruction is a very efficient way to pass AL as a four-byte parameter object. On the other hand, if you decide to pass the AH register as the byte parameter, pushing EAX won't work because this would leave the value in AH at offset EBP+9 in the activation record shown in Figure 3.9. Unfortunately, the procedure expects this value at offset EBP+8 so simply pushing EAX won't do the job. If you pass AH, BH, CH, or DH as a byte parameter, HLA emits code like the following:

```
sub( 4, esp );    // Make room for the parameter on the stack.
mov( ah, [esp] ); // Store AH into the L.O. byte of the parameter.
```

As you can clearly see, passing one of the "H" registers as a byte parameter is less efficient (two instructions) than passing one of the "L" registers. So you should attempt to use the "L" registers whenever possible if passing an eight-bit register as a parameter[11]. Note, by the way, that there is very little you can do about the difference in efficiency, even if you manually pass the parameters yourself.

If the byte parameter you decide to pass is a variable rather than a register, HLA generates decidedly worse code. For example, suppose you call OneByteParm as follows:

```
OneByteParm( uns8Var );
```

For this call, HLA will emit code similar to the following to push this single byte parameter:

```
push( eax );
push( eax );
mov( uns8Var, al );
mov( al, [esp+4] );
```

11. Or better yet, pass the parameter directly in the register if you are writing the procedure yourself.

```
pop( eax );
```

As you can plainly see, this is a lot of code to pass a single byte on the stack! HLA emits this much code because (1) it guarantees not to disturb any registers, and (2) it doesn't know whether *uns8Var* is the last variable in allocated memory. You can generate much better code if you don't have to enforce either of these two constraints.

If you've got a spare 32-bit register laying around (especially one of EAX, EBX, ECX or EDX) then you can pass a byte parameter on the stack using only two instructions. Move (or move with zero/sign extension) the byte value into the register and then push the register onto the stack. For the current call to *OneByteParm*, the calling sequence would look like the following in EAX is available:

```
mov( uns8Var, al );
push( eax );
call OneByteParm;
```

If only ESI or EDI were available, you could use code like this:

```
movzx( uns8Var, esi );
push( esi );
call OneByteParm;
```

Another trick you can use to pass the parameter with only a single push instruction is to coerce the byte variable to a double word object, i.e.,

```
push( (type dword uns8Var));
call OneByteParm;
```

This last example is very efficient. Note that it pushes the first three bytes of whatever value happens to follow *uns8Var* in memory as the padding bytes. HLA doesn't use this technique because there is a (very tiny) chance that using this scheme will cause the program to fail. If it turns out that the *uns8Var* object is the last byte of a given page in memory and the next page of memory is unreadable, the PUSH instruction will cause a memory access exception. To be on the safe side, the HLA compiler does not use this scheme. However, if you always ensure that the actual parameter you pass in this fashion is not the last variable you declare in a static section, then you can get away with code that uses this technique. Since it is nearly impossible for the byte object to appear at the last accessible address on the stack, it is probably safe to use this technique with VAR objects.

When passing word parameters on the stack you must also ensure that you include padding bytes so that each parameter consumes an even multiple of four bytes. You can use the same techniques we use to pass bytes except, of course, there are two valid bytes of data to pass instead of one. For example, you could use either of the following two schemes to pass a word object *w* to a *OneWordParm* procedure:

```
mov( w, ax );
push( eax );
call OneWordParm;


push( (type dword w) );
call OneWordParm;
```

When passing large objects by value on the stack (e.g., records and arrays), you do not have to ensure that each element or field of the object consumes an even multiple of four bytes; all you need to do is ensure that the entire data structure consumes an even multiple of four bytes on the stack. For example, if you have an array of 10 three-byte elements, the entire array will need two bytes of padding (10*3 is 30 bytes which is not evenly divisible by four, but 10*3 + 2 is 32 which is divisible by four). HLA does a fairly good job of passing large data objects by value to a procedure. For larger objects, you should use the HLA high level language procedure invocation syntax unless you have some special requirements. Of course, if you want efficient operation, you should try to avoid passing large data structures by value.

By default, HLA guarantees that it won't disturb the values of any registers when it emits code to pass parameters to a procedure. Sometimes this guarantee isn't necessary. For example, if you are returning a

© 2001, By Randall Hyde

function result in EAX and you are not passing a parameter to a procedure in EAX, there really is no reason to preserve EAX upon entry into the procedure. Rather than generating some crazy code like the following to pass a byte parameter:

```
push( eax );
push( eax );
mov( uns8Var, al );
mov( al, [esp+4] );
pop( eax );
```

HLA could generate much better code if it knows that it can use EAX (or some other register):

```
mov( uns8Var, al );
push( eax );
```

You can use the @USE procedure option to tell HLA that it can modify a register's value if doing so would improve the code it generates when passing parameters. The syntax for this option is

$$\texttt{@use genReg}_{32}\texttt{;}$$

The $genReg_{32}$ operand can be EAX, EBX, ECX, EDX, ESI, or EDI. You'll obtain the best results if this register is one of EAX, EBX, ECX, or EDX. Particularly, you should note that you cannot specify EBP or ESP here (since the procedure already uses those registers).

The @USE procedure option tells HLA that it's okay to modify the value of the register you specify as an operand. Therefore, if HLA can generate better code by not preserving that register's value, it will do so. For example, when the "@use eax;" option is provided for the *OneByteParm* procedure given earlier, HLA will only emit the two instructions immediately above rather than the five-instruction sequence that preserves EAX.

You must exercise care when specifying the @USE procedure option. In particular, you should not be passing any parameters in the same register you specify in the @USE option (since HLA may inadvertently scramble the parameter's value if you do this). Likewise, you must ensure that it's really okay for the procedure to change the register's value. As noted above, the best choice for an @USE register is EAX when the procedure is returning a function result in EAX (since, clearly, the caller will not expect the procedure to preserve EAX).

If your procedure has a FORWARD or EXTERNAL declaration, the @USE option must only appear in the FORWARD or EXTERNAL definition, not in the actual procedure declaration. If no such procedure prototype appears, then you must attach the @USE option to the procedure declaration.

Example:

```
procedure OneByteParm( b:byte ); @nodisplay; @use EAX;
begin OneByteParm;

    << Do something with b >>

end OneByteParm;
    .
    .
    .
static
    byteVar:byte;
        .
        .
        .
    OneByteParm( byteVar );
```

This call to OneByteParm emits the following instructions:

```
mov( uns8Var, al );
push( eax );
```

3.8.5.3 Accessing Reference Parameters on the Stack

Since HLA passes the address of the actual parameters for reference parameters, accessing the reference parameters within a procedure is slightly more difficult than accessing value parameters because you have to dereference the pointers to the reference parameters. Unfortunately, HLA's high level syntax for procedure declarations and invocations does not (and cannot) abstract this detail away for you. You will have to manually dereference these pointers yourself. This section reviews how you do this.

Consider the following program:

```
program AccessingReferenceParameters;
#include( "stdlib.hhf" )

    procedure RefParm( var theParameter: uns32 ); @nodisplay;
    begin RefParm;

        // Add two directly to the parameter passed by
        // reference to this procedure.

        mov( theParameter, eax );
        add( 2, (type uns32 [eax]) );

        // Fetch the value of the reference parameter
        // and print it's value.

        mov( [eax], eax );
        stdout.put
        (
            "theParameter now equals ",
            (type uns32 eax),
            nl
        );

    end RefParm;


static
    p1: uns32 := 10;
    p2: uns32 := 15;

begin AccessingReferenceParameters;

    RefParm( p1 );
    RefParm( p2 );

    stdout.put( "On return, p1=", p1, " and p2=", p2, nl );

end AccessingReferenceParameters;
```

Program 3.7 Accessing a Reference Parameter

 © 2001, By Randall Hyde

In this example the *RefParm* procedure has a single pass by reference parameter. Pass by reference parameters are always a pointer to the type specified by the parameter's declaration. Therefore, *theParameter* is actual an object of type "pointer to uns32" rather than an *uns32* value. In order to access the value associated with *theParameter*, this code has to load that double word address into a 32-bit register and access the data indirectly. The "mov(theParameter, eax);" instruction in the code above fetches this pointer into the EAX register and then the procedure uses the "[eax]" addressing mode to access the actual value of *theParameter.*

Since this procedure accesses the data of the actual parameter, adding two to this data affects the values of the variables passed to the *RefParm* procedure from the main program. Of course, this should come as no surprise since this is the standard semantics for pass by reference parameters.

As you can see, accessing (small) pass by reference parameters is a little less efficient than accessing value parameters because you need an extra instruction to load the address into a 32-bit pointer register (not to mention, you have to reserve a 32-bit register for this purpose). If you access reference parameters frequently, these extra instructions can really begin to add up, reducing the efficiency of your program. Furthermore, it's easy to forget to dereference a reference parameter and use the address of the value instead of the value in your calculations (this is especially true when passing double-word parameters, like the *uns32* parameter in the example above, to your procedures). Therefore, unless you really need to affect the value of the actual parameter, you should use pass by value to pass small objects to a procedure.

Passing large objects, like arrays and records, is where reference parameters become very efficient. When passing these objects by value, the calling code has to make a copy of the actual parameter; if the actual parameter is a large object, the copy process can be very inefficient. Since computing the address of a large object is just as efficient as computing the address of a small scalar object, there is no efficiency loss when passing large objects by reference. Within the procedure you must still dereference the pointer to access the object but the efficiency loss due to indirection is minimal when you contrast this with the cost of copying that large object. The following program demonstrates how to use pass by reference to initialize an array of records:

```
program accessingRefArrayParameters;
#include( "stdlib.hhf" )

const
    NumElements := 64;

type
    Pt: record

            x:uns8;
            y:uns8;

        endrecord;

    Pts: Pt[NumElements];


    procedure RefArrayParm( var ptArray: Pts ); @nodisplay;
    begin RefArrayParm;

        push( eax );
        push( ecx );
        push( edx );

        mov( ptArray, edx );    // Get address of parameter into EDX.

        for( mov( 0, ecx ); ecx < NumElements; inc( ecx )) do

            // For each element of the array, set the "x" field
```

```
                // to (ecx div 8) and set the "y" field to (ecx mod 8).

                mov( cl, al );
                shr( 3, al );   // ECX div 8.
                mov( al, (type Pt [edx+ecx*2]).x );

                mov( cl, al );
                and( %111, al );  // ECX mod 8.
                mov( al, (type Pt [edx+ecx*2]).y );

            endfor;
            pop( edx );
            pop( ecx );
            pop( eax );

        end RefArrayParm;


    static
        MyPts: Pts;

    begin accessingRefArrayParameters;

        // Initialize the elements of the array.

        RefArrayParm( MyPts );


        // Display the elements of the array.

        for( mov( 0, ebx ); ebx < NumElements; inc( ebx )) do

            stdout.put
            (
                "RefArrayParm[",
                (type uns32 ebx):2,
                "].x=",
                MyPts.x[ ebx*2 ],

                "   RefArrayParm[",
                (type uns32 ebx):2,
                "].y=",
                MyPts.y[ ebx*2 ],
                nl
            );

        endfor;

    end accessingRefArrayParameters;
```

Program 3.8 Passing an Array of Records by Referencing

As you can see from this example, passing large objects by reference isn't particularly inefficient. Other than tying up the EDX register throughout the *RefArrayParm* procedure plus a single instruction to load EDX with the address of the reference parameter, the *RefArrayParm* procedure doesn't require many more instructions than the same procedure where you would pass the parameter by value.

 © 2001, By Randall Hyde

3.8.5.4 Passing Reference Parameters on the Stack

HLA's high level syntax often makes passing reference parameters a breeze. All you need to do is specify the name of the actual parameter you wish to pass in the procedure's parameter list. HLA will automatically emit some code that will compute the address of the specified actual parameter and push this address onto the stack. However, like the code HLA emits for value parameters, the code HLA generates to pass the address of the actual parameter on the stack may not be the most efficient that is possible. Therefore, if you want to write fast code, you may want to manually write the code to pass reference parameters to a procedure. This section discusses how to do exactly that.

Whenever you pass a static object as a reference parameter, HLA generates very efficient code to pass the address of that parameter to the procedure. As an example, consider the following code fragment:

```
procedure HasRefParm( var d:dword );
    .
    .
    .
static
    FourBytes:dword;

var
    v: dword;
    .
    .
    .
HasRefParm( FourBytes );
    .
    .
    .
```

For the call to the *HasRefParm* procedure, HLA emits the following instruction sequence:

```
    pushd( &FourBytes );
    call HasRefParm;
```

You really aren't going to be able to do substantially better than this if you are passing your reference parameters on the stack. So if you're passing static objects as reference parameters, HLA generates fairly good code and you should stick with the high level syntax for the procedure call.

Unfortunately, when passing automatic (VAR) objects or indexed variables as reference parameters, HLA needs to compute the address of the object at run-time. This generally requires the use of the LEA instruction. Unfortunately, the LEA instruction requires the use of a 32-bit register and HLA promises not to disturb the values in any registers when it automatically generates code for you[12]. Therefore, HLA needs to preserve the value in whatever register it uses when it computes an address via LEA to pass a parameter by reference. The following example shows you the code that HLA actually emits:

```
//  Call to the HasRefParm procedure:

        HasRefParm( v );

// HLA actually emits the following code for the above call:

        push( eax );
        push( eax );
        lea( eax, v );
        mov( eax, [esp+4] );
        pop( eax );
```

12. This isn't entirely true. You'll see the exception in the chapter on Classes and Objects. Also, using the @USE procedure option tells HLA that it's okay to modify the value in one of the registers.

```
call HasRefParm;
```

As you can see, this is quite a bit of code, especially if you have a 32-bit register available and you don't need to preserve that register's value. Here's a better code sequence given the availability of EAX:

```
lea( eax, v );
push( eax );
call HasRefParm;
```

Remember, when passing an actual parameter by reference, you must compute the address of that object and push the address onto the stack. For simple static objects you can use the address-of operator ("&") to easily compute the address of the object and push it onto the stack; however, for indexed and automatic objects, you will probably need to use the LEA instruction to compute the address of the object. Here are some examples that demonstrate this using the *HasRefParm* procedure from the previous examples:

```
static
    i:    int32;
    Ary:  int32[16];
    iptr: pointer to int32 := &i;

var
    v:    int32;
    AV:   int32[10];
    vptr: pointer to int32;
     .
     .
     .
    lea( eax, v );
    mov( eax, vptr );
     .
     .
     .
// HasRefParm( i );

    push( &i );        // Simple static object, so just use "&".
    call HasRefParm;

// HasRefParm( Ary[ebx] );   // Pass element of Ary by reference.

    lea( eax, Ary[ ebx*4 ]);  // Must use LEA for indexed addresses.
    push( eax );
    call HasRefParm;

// HasRefParm( *iptr );  -- Pass object pointed at by iptr

    push( iptr );             // Pass address (iptr's value) on stack.
    call HasRefParm;

// HasRefParm( v );

    lea( eax, v );            // Must use LEA to compute the address
    push( eax );              //  of automatic vars passed on stack.
    call HasRefParm;

// HasRefParm( AV[ esi ] );  -- Pass element of AV by reference.

    lea( eax, AV[ esi*4] );   // Must use LEA to compute address of the
    push( eax );              //  desired element.
    call HasRefParm;

// HasRefParm( *vptr );  -- Pass address held by vptr...
```

 © 2001, By Randall Hyde

```
push( vptr );              // Just pass vptr's value as the specified
call HasRefParm;           //  address.
```

If you have an extra register to spare, you can tell HLA to use that register when computing the address of reference parameters (without emitting the code to preserve that register's value). The @USE option will tell HLA that it's okay to use the specified register without preserving it's value. As noted in the section on value parameters, the syntax for this procedure option is

```
@use reg32;
```

where reg_{32} may be any of EAX, EBX, ECX, EDX, ESI, or EDI. Since reference parameters always pass a 32-bit value, all of these registers are equivalent as far as HLA is concerned (unlike value parameters, that may prefer the EAX, EBX, ECX, or EDX register). Your best choice would be EAX if the procedure is not passing a parameter in the EAX register and the procedure is returning a function result in EAX; otherwise, any currently unused register will work fine.

With the "@USE EAX;" option, HLA emits the shorter code given in the previous examples. It does not emit all the extra instructions needed to preserve EAX's value. This makes your code much more efficient, especially when passing several parameters by reference or when calling procedures with reference parameters several times.

3.8.5.5 Passing Formal Parameters as Actual Parameters

The examples in the previous two sections show how to pass static and automatic variables as parameters to a procedure, either by value or by reference. There is one situation that these sections don't handle properly: the case when you are passing a formal parameter in one procedure as an actual parameter to another procedure. The following simple example demonstrates the different cases that can occur for pass by value and pass by reference parameters:

```
procedure p1( val v:dword;  var r:dword );
begin p1;
     .
     .
     .
end p1;

procedure p2( val v2:dword; var r2:dword );
begin p2;

    p1( v2, r2 );    // (1) First call to p1.
    p1( r2, v2 );    // (2) Second call to p1.

end p2;
```

In the statement labelled (1) above, procedure *p2* calls procedure *p1* and passes its two formal parameters as parameters to *p1*. Note that this code passes the first parameter of both procedures by value and it passes the second parameter of both procedures by reference. Therefore, in statement (1), the program passes the *v2* parameter into *p2* by value and passes it on to *p1* by value; likewise, the program passes *r2* in by reference and it passes the value onto *p2* by reference.

Since *p2*'s caller passes *v2* in by value and *p2* passes this parameter to *p1* by value, all the code needs to do is make a copy of *v2*'s value and pass this on to *p1*. The code to do this is nothing more than a single push instruction, e.g.,

```
push( v2 );
<< code to handle r2 >>
call p1;
```

As you can see, this code is identical to passing an automatic variable by value. *Indeed, it turns out that the code you need to write to pass a value parameter to another procedure is identical to the code you would write to pass a local, automatic, variable to that other procedure.*

Passing *r2* in statement (1) above requires a little more thought. You do not take the address of *r2* using the LEA instruction as you would a value parameter or an automatic variable. When passing *r2* on through to *p1*, the author of this code probably expects the *r* formal parameter to contain the address of the variable whose address *p2*'s caller passed into *p2*. In plain English, this means that *p2* must pass the address of *r2*'s actual parameter on through to *p1*. Since the *r2* parameter is actually a double word value containing the address of the corresponding actual parameter, this means that the code must pass the dword value of *r2* on to *p1*. The complete code for statement (1) above looks like the following:

```
push( v2 );   // Pass the value passed in through v2 to p1.
push( r2 );   // Pass the address passed in through r2 to p1.
call p1;
```

The important thing to note in this example is that passing a formal reference parameter (*r2*) as an actual reference parameter (*r*) does not involve taking the address of the formal parameter (*r2*). *P2*'s caller has already done this; *p2* need only pass this address on through to *p1*.

In the second call to *p1* in the example above (2), the code swaps the actual parameters so that the call to *p1* passes *r2* by value and *v2* by reference. Specifically, *p1* expects *p2* to pass it the value of the dword object associated with *r2*; likewise, it expects *p2* to pass it the address of the value associated with *v2*.

To pass the value of the object associated with *r2*, your code must dereference the pointer associated with *r2* and directly pass the value. Here is the code HLA automatically generates to pass *r2* as the first parameter to *p1* in statement (2):

```
sub( 4, esp );      // Make room on stack for parameter.
push( eax );        // Preserve EAX's value.
mov( r2, eax );     // Get address of object passed in to p2.
mov( [eax], eax );  // Dereference to get the value of this object.
mov( eax, [esp+4]);// Put value of parameter into its location on stack.
pop( eax );         // Restore original EAX value.
```

As usual, HLA generates a little more code than may be necessary because it won't destroy the value in the EAX register (you may use the @USE procedure option to tell HLA that it's okay to use EAX's value, thereby reducing the code it generates). You can write more efficient code if a register is available to use in this sequence. If EAX is unused, you could trim this down to the following:

```
mov( r2, eax );     // Get the pointer to the actual object.
pushd( [eax] );     // Push the value of the object onto the stack.
```

Since you can treat value parameters exactly like local (automatic) variables, you use the same code to pass *v2* by reference to *p1* as you would to pass a local variable in *p2* to *p1*. Specifically, you use the LEA instruction to compute the address of the value in the *v2*. The code HLA automatically emits for statement (2) above preserves all registers and takes the following form (same as passing an automatic variable by reference):

```
push( eax );        // Make room for the parameter.
push( eax );        // Preserve EAX's value.
lea( eax, v2 );     // Compute address of v2's value.
mov( eax, [esp+4]);// Store away address as parameter value.
pop( eax );         // Restore EAX's value
```

Of course, if you have a register available, you can improve on this code. Here's the complete code that corresponds to statement (2) above:

```
mov( r2, eax );     // Get the pointer to the actual object.
pushd( [eax] );     // Push the value of the object onto the stack.
lea( eax, v2 );     // Push the address of V2 onto the stack.
push( eax );
call p1;
```

 © 2001, By Randall Hyde

3.8.5.6 HLA Hybrid Parameter Passing Facilities

Like control structures, HLA provides a high level language syntax for procedure calls that is convenient to use and easy to read. However, this high level language syntax is sometimes inefficient and may not provide the capabilities you need (for example, you cannot specify an arithmetic expression as a value parameter as you can in high level languages). HLA lets you overcome these limitations by writing low-level ("pure") assembly language code. Unfortunately, the low-level code is harder to read and maintain than procedure calls that use the high level syntax. Furthermore, it's quite possible that HLA generates perfectly fine code for certain parameters and only one or two parameters present a problem. Fortunately, HLA provides a hybrid syntax for procedure calls that allows you to use both high-level and low-level syntax as appropriate for a given actual parameter. This lets you use the high level syntax where appropriate and then drop down into pure assembly language to pass those special parameters that HLA's high level language syntax cannot handle efficiently (if at all).

Within an actual parameter list (using the high level language syntax), if HLA encounters "#{" followed by a sequence of statements and a closing "}#", HLA will substitute the instructions between the braces in place of the code it would normally generate for that parameter. For example, consider the following code fragment:

```
procedure HybridCall( i:uns32; j:uns32 );
begin HybridCall;
    .
    .
    .
end HybridCall;
     .
     .
     .

    // Equivalent to HybridCall( 5, i+j );

    HybridCall
    (
        5,
        #{
            mov( i, eax );
            add( j, eax );
            push( eax );
        }#
    );
```

The call to *HybridCall* immediately above is equivalent to the following "pure" assembly language code:

```
pushd( 5 );
mov( i, eax );
add( j, eax );
push( eax );
call HybridCall;
```

As a second example, consider the example from the previous section:

```
procedure p2( val v2:dword; var r2:dword );
begin p2;

    p1( v2, r2 );    // (1) First call to p1.
    p1( r2, v2 );    // (2) Second call to p1.
```

```
end p2;
```

HLA generates exceedingly mediocre code for the second call to *p1* in this example. If efficiency is important in the context of this procedure call, and you have a free register available, you might want to rewrite this code as follows[13]:

```
procedure p2( val v2:dword; var r2:dword );
begin p2;

    p1( v2, r2 );    // (1) First call to p1.
    p1               // (2) Second call to p1.
    (                //     This code assumes EAX is free.
        #{
            mov( r2, eax );
            pushd( [eax] );
        }#,

        #{
            lea( eax, v2 );
            push( eax );
        }#
    );

end p2;
```

3.8.5.7 Mixing Register and Stack Based Parameters

You can mix register parameters and standard (stack-based) parameters in the same high level procedure declaration, e.g.,

```
procedure HasBothRegAndStack( var dest:dword in edi; count:un32 );
```

When constructing the activation record, HLA ignores the parameters you pass in registers and only processes those parameters you pass on the stack. Therefore, a call to the *HasBothRegAndStack* procedure will push only a single parameter onto the stack (*count*). It will pass the *dest* parameter in the EDI register. When this procedure returns to its caller, it will only remove four bytes of parameter data from the stack.

Note that when you pass a parameter in a register, you should avoid specifying that same register in the @USE procedure option. In the example above, HLA might not generate any code whatsoever at all for the *dest* parameter (because the value is already in EDI). Had you specified "@use edi;" and HLA decided it was okay to disturb EDI's value, this would destroy the parameter value in EDI; that won't actually happen in this particular example (since HLA never uses a register to pass a dword value parameter like *count*), but keep this problem in mind.

3.9 Procedure Pointers

The x86 CALL instruction is very similar to the JMP instruction. In particular, it allows the same three basic forms as the JMP instruction: direct calls (to a procedure name), indirect calls through a 32-bit general

13. Of course, you could also use the "@use eax;" procedure option to achieve the same effect in this example.

purpose register, and indirect calls through a double word pointer variable. The CALL instruction allows the following (low-level) syntax supporting these three types of procedure invocations:

```
call Procname;    // Direct call to procedure "Procname" (or stmt label).
call( Reg32 );     // Indirect call to procedure whose address appears
                  //   in the Reg32 general-purpose 32-bit register.
call( dwordVar ); // Indirect call to the procedure whose address appears
                  //   in the dwordVar double word variable.
```

HLA treats procedure names like static objects. Therefore, you can compute the address of a procedure by using the address-of ("&") operator along with the procedure's name or by using the LEA instruction. For example, "&Procname" is the address of the very first instruction of the *Procname* procedure. Therefore, all three of the following code sequences wind up calling the *Procname* procedure:

```
call Procname;
  .
  .
  .
mov( &Procname, eax );
call( eax );
  .
  .
  .
lea( eax, Procname );
call( eax );
```

Since the address of a procedure fits in a 32-bit object, you can store such an address into a dword variable; in fact, you can initialize a dword variable with the address of a procedure using code like the following:

```
    procedure p;
    begin p;
    end p;
      .
      .
      .
static
    ptrToP: dword := &p;
      .
      .
      .
    call( ptrToP );  // Calls the "p" procedure if ptrToP has not changed.
```

Because the use of procedure pointers occurs frequently in assembly language programs, HLA provides a special syntax for declaring procedure pointer variables and for calling procedures indirectly through such pointer variables. To declare a procedure pointer in an HLA program, you can use a variable declaration like the following:

```
static
    procPtr: procedure;
```

Note that this syntax uses the keyword PROCEDURE as a data type. It follows the variable name and a colon in one of the variable declaration sections (STATIC, READONLY, STORAGE, or VAR). This sets aside exactly four bytes of storage for the *procPtr* variable. To call the procedure whose address is held by *procPtr*, you can use either of the following two forms:

```
call( procPtr );    // Low-level syntax.
procPtr();          // High-level language syntax.
```

Note that the high level syntax for an indirect procedure call is identical to the high level syntax for a direct procedure call. HLA can figure out whether to use a direct call or an indirect call by the type of the identi-

 © 2001, By Randall Hyde

fier. If you've specified a variable name, HLA assumes it needs to use an indirect call; if you specify a procedure name, HLA uses a direct call.

Like all pointer objects, you should not attempt to indirectly call a procedure through a pointer variable unless you've initialized that variable with the address appropriately. There are two ways to initialize a procedure pointer variable: STATIC and READONLY objects allow an initializer, or you can compute the address of a routine (as a 32-bit value) and store that 32-bit address directly into the procedure pointer at run-time. The following code fragment demonstrates both ways you can initialize a procedure pointer.

```
static
    ProcPtr: procedure := &p;    // Initialize ProcPtr with the address of p.
     .
     .
     .
    ProcPtr();             // First invocation calls p.

    mov( &q, ProcPtr );    // Reload ProcPtr with the address of q.
     .
     .
     .
    ProcPtr();             // This invocation calls the "q" procedure.
```

Procedure pointer variable declarations also allow the declaration of parameters. To declare a procedure pointer with parameters, you must use a declaration like the following:

```
static
    p:procedure( i:int32; c:char );
```

This declaration states that *p* is a 32-bit pointer that contains the address of a procedure having two parameters. If desired, you could also initialize this variable *p* with the address of some procedure by using a static initializer, e.g.,

```
static
    p:procedure( i:int32; c:char ) := &SomeProcedure;
```

Note that *SomeProcedure* must be a procedure whose parameter list exactly matches *p*'s parameter list (i.e., two value parameters, the first is an *int32* parameter and the second is a *char* parameter). To indirectly call this procedure, you could use either of the following sequences:

```
    push( << Value for i >> );
    push( << Value for c >> );
    call( p );
-or-
    p( <<Value for i>>, <<Value for c>> );
```

The high level language syntax has the same features and restrictions as the high level syntax for a direct procedure call. The only difference is the actual CALL instruction HLA emits at the end of the calling sequence.

Although all of the examples in this section have used STATIC variable declarations, don't get the idea that you can only declare simple procedure pointers in the STATIC or other variable declaration sections. You can declare procedure pointer types in the TYPE section. You can declare procedure pointers as fields of a RECORD. Assuming you create a type name for a procedure pointer in the TYPE section, you can even create arrays of procedure pointers. The following code fragments demonstrate some of the possibilities:

```
type
    pptr:   procedure;
    prec:   record
                p:pptr;
                // other fields...
            endrecord;
static
    p1:pptr;
```


© 2001, By Randall Hyde

```
p2:pptr[2]
p3:prec;
  .
  .
  .
p1();
p2[ebx*4]();
p3.p();
```

One very important thing to keep in mind when using procedure pointers is that HLA does not (and cannot) enforce strict type checking on the pointer values you assign to a procedure pointer variable. In particular, if the parameter lists do not agree between the declarations of the pointer variable and the procedure whose address you assign to the pointer variable, the program will probably crash if you attempt to call the mismatched procedure indirectly through the pointer using the high level syntax. Like the low-level "pure" procedure calls, it is your responsibility to ensure that the proper number and types of parameters are on the stack prior to the call.

3.10 Procedural Parameters

One place where procedure pointers are quite invaluable is in parameter lists. Selecting one of several procedures to call by passing the address of some procedure, selected from a set of procedures, is not an uncommon operation. Therefore, HLA lets you declare procedure pointers as parameters.

There is nothing special about a procedure parameter declaration. It looks exactly like a procedure variable declaration except it appears within a parameter list rather than within a variable declaration section. The following are some typical procedure prototypes that demonstrate how to declare such parameters:

```
procedure p1( procparm: procedure ); forward;
procedure p2( procparm: procedure( i:int32 ) ); forward;
procedure p3( val procparm: procedure ); forward;
```

The last example above is identical to the first. It does point out, though, that you generally pass procedural parameters by value. This may seem counter-intuitive since procedure pointers are addresses and you will need to pass an address as the actual parameter; however, a pass by reference procedure parameter means something else entirely. consider the following (legal!) declaration:

```
procedure p4( var procPtr:procedure ); forward;
```

This declaration tells HLA that you are passing a procedure *variable* by reference to *p4*. The address HLA expects must be the address of a procedure pointer variable, not a procedure.

When passing a procedure pointer by value, you may specify either a procedure variable (whose value HLA passes to the actual procedure) or a procedure pointer constant. A procedure pointer constant consists of the address-of operator ("&") immediately followed by a procedure name. Passing procedure constants is probably the most convenient way to pass procedural parameters. For example, the following calls to the *Plot* routine might plot out the function passed as a parameter from -2π to $+2\pi$.

```
Plot( &sineFunc );
Plot( &cosFunc  );
Plot( &tanFunc  );
```

Note that you cannot pass a procedure as a parameter by simply specifying the procedure's name. I.e., "Plot(sineFunc);" will not work. Simply specifying the procedure name doesn't work because HLA will attempt to directly call the procedure whose name you specify (remember, a procedure name inside a parameter list invokes instruction composition). However, since you don't specify a parameter list, or at least an empty pair of parentheses, after the parameter/procedure's name, HLA generates a syntax error message. Moral of the story: don't forget to preface procedure parameter constant names with the address-of operator.

 © 2001, By Randall Hyde

3.11 Untyped Reference Parameters

Sometimes you will want to write a procedure to which you pass a generic memory object by reference without regard to the type of that memory object. A classic example is a procedure that zeros out some data structure. Such a procedure might have the following prototype:

```
procedure ZeroMem( var mem:byte; count:uns32 );
```

This procedure would zero out *count* bytes starting at the address the first parameter specifies. The problem with this procedure prototype is that HLA will complain if you attempt to pass anything other than a byte object as the first parameter. Of course, you can overcome this problem using type coercion like the following, but if you call this procedure several times with lots of different data types, then the following coercion operator is rather tedious to use:

```
ZeroMem( (type byte MyDataObject), @size( MyDataObject ));
```

Of course, you can always use hybrid parameter passing or manually push the parameters yourself, but these solutions are even more work than using the type coercion operation. Fortunately, HLA provides a far more convenient solution: untyped reference parameters.

Untyped reference parameters are exactly that – pass by reference parameters on which HLA doesn't bother to compare the type of the actual parameter against the type of the formal parameter. With an untyped reference parameter, the call to *ZeroMem* above would take the following form:

```
ZeroMem( MyDataObject, @size( MyDataObject ));
```

MyDataObject could be any type and multiple calls to *ZeroMem* could pass different typed objects without any objections from HLA.

To declare an untyped reference parameter, you specify the parameter using the normal syntax except that you use the reserved word VAR in place of the parameter's type. This VAR keyword tells HLA that any variable object is legal for that parameter. Note that you must pass untyped reference parameters by reference, so the VAR keyword must precede the parameter's declaration as well. Here's the correct declaration for the *ZeroMem* procedure using an untyped reference parameter:

```
procedure ZeroMem( var mem:var; count:uns32 );
```

With this declaration, HLA will compute the address of whatever memory object you pass as an actual parameter to *ZeroMem* and pass this on the stack.

3.12 Iterators and the FOREACH Loop

One nifty feature HLA provides is support for *true* iterators[14]. An iterator is a special type of procedure or function that you use in conjunction with the HLA FOREACH..ENDFOR loop. Combined, these two language features (iterators and the FOREACH..ENDFOR loop) provide a very powerful user-defined looping construct.

The HLA FOREACH..ENDFOR statement uses the following basic syntax:

```
foreach iteratorID( optional_parameters ) do

    << loop body >>
```

14. HLA's iterators are based on the control structure by the same name from the CLU programming language. Those things that C/C++ programmers refer to as iterators are more properly called *cursors*. While it is certainly possible to write cursors in HLA, it is important to note that HLA's iterators are quite a bit more powerful than C/C++'s iterators.

```
endfor;
```

The FOREACH statement calls the specified iterator. If the iterator *succeeds*, then the FOREACH statement executes the loop body; if the iterator *fails*, then control transfers to the first statement following the ENDFOR clause. On each iteration of the loop body, the program re-enters the iterator code and, once again, the iterator returns success or failure to determine whether to repeat the loop body.

At first glance, you might get the impression that the FOREACH loop is nothing more than a WHILE loop and an iterator is a function that returns true (success) or false (failure). However, this is not an accurate picture of how the FOREACH loop operates. First of all, the FOREACH loop does not CALL the iterator on each iteration of the loop; it *re-enters* the iterator. Specifically, control does not (necessarily) begin with the first statement of the iterator whenever control returns to the top of the FOREACH loop. The second big difference between a FOREACH/iterator loop and a WHILE/function loop is that the iterator procedure maintains its activation record in memory for the duration of the FOREACH loop. A function you would call from a WHILE loop, by contrast, builds and destroys the function's activation record on each iteration of the loop. This means that the iterator's local (automatic) variables maintain their values until the FOREACH loop terminates. This has important ramifications, especially for recursive iterator functions.

An iterator declaration looks very similar to a procedure declaration. Indeed, about the only syntactical difference is the use of the reserved word ITERATOR rather than PROCEDURE. The following is an example of a simple iterator:

```
iterator range( start:uns32;  last:uns32 ); nodisplay;
begin range;

    mov( start, eax );
    while( eax <= last ) do

        push( eax );
        yield();
        pop( eax );
        inc( eax );

    endwhile;

end range;
```

The only thing special about this iterator declaration, other than the use of the ITERATOR reserved word, is that it calls a special procedure named *yield*. In a few paragraphs you'll see the purpose of the call to the *yield* procedure.

A typical FOREACH loop that calls the *range* iterator might look like the following:

```
foreach range( 1, 10 ) do

    stdout.put( "Iteration = ", (type uns32 eax), nl );

endfor;
```

Here's how the iterator and the FOREACH loop work together. Upon first encountering the FOREACH statement, the program makes an *initial call* to the *range* iterator. Except for a few extra parameters HLA pushes on the stack, this call is exactly like a standard procedure call. Upon entry into the iterator, the *start* parameter has the initial value one and the *last* parameter has the initial value ten. The iterator loads *start* into EAX and compares this against the value in *last* (ten). Since EAX's value is less than or equal to ten, the program enters the loop's body. The loop body pushes EAX's value onto the stack and then calls the *yield* procedure. The *yield* procedure transfers control to the body of the FOREACH loop that called the *range* iterator in the first place. Calling *yield* is how the iterator returns success to the FOREACH loop. Within the body of the FOREACH loop, above, the code prints out the value of the EAX register as an unsigned integer. During the first iteration of the loop, EAX contains one so the loop body prints this value.

 © 2001, By Randall Hyde

At the bottom of the FOREACH loop, the program *re-enters* the iterator. When the FOREACH loop re-enters the iterator, it transfers control to the first statement following the call to the *yield* function. Intuitively, you can view the FOREACH loop body as a procedure that the iterator calls whenever you call the *yield* function[15]. Whenever the program encounters the ENDFOR clause, it returns to the iterator, executing the first statement beyond the *yield* call. In the current example, this pops the value of EAX off the stack (preserved before the call to *yield*), the loop increments EAX and repeats as long as EAX is less than ten.

When the *range* iterator increments EAX to 11, the WHILE loop in the iterator terminates and control falls off the bottom of the iterator. This is how an iterator returns failure to the calling FOREACH loop. At that point control transfers to the first statement following the ENDFOR in the FOREACH..ENDFOR loop.

By the way, the *range* iterator, combined with the FOREACH loop above, creates a relatively inefficient implementation of the following loop:

```
for( mov( 1, eax ); eax < 10; inc( eax )) do

    stdout.put( "Iteration = ", (type uns32 eax), nl );

endfor;
```

However, don't get the impression from this example that iterators are particularly inefficient. Iterators are not a good choice for something like *range*. However, there are many iterators you can write that are just as efficient as other means of loop control and computation.

An important point to remember when using iterators is that the iterator's activation record remains on the stack as long as the iterator returns success. The program only removes the activation record when the iterator fails. The *range* iterator takes advantage of this fact since it refers to the value of its *last* parameter on each re-entry from the FOREACH loop. The fact that parameters and local (automatic) variables maintain their values for the duration of the FOREACH loop is very important to many algorithms that use iterators, especially recursive algorithms.

One side effect of having an iterator maintain its activation record until it fails is that the value of ESP changes considerably between the statement immediately before the FOREACH statement and the first statement in the body of the FOREACH loop. This is because the program "pushes" the activation record onto the stack upon encountering the FOREACH loop and doesn't "pop" this activation record off the stack until the FOREACH loop fails. Therefore, code like the following will not work as expected:

```
pushd( 10 );
foreach range( 1, 25 ) do
    pop( ebx );
    push( ebx );
    stdout.put( "eax=", eax, " ebx=", ebx, nl );
endfor;
pop( ebx );
```

The problem with this code is that the FOREACH loop pushes a whole lot of data onto the stack after the PUSHD instruction pushes the value 10 onto the stack. Therefore, the POP instruction inside the loop does not pop the value 10 from the stack. Instead, it pops some data pushed on the stack by the iterator (specifically, it pops the return address that transfers control to the first instruction following the *yield* call). Therefore, you cannot use the stack to transfer data into or out of a FOREACH loop[16].

Another problem with the stack and the FOREACH loop occurs if you try to prematurely exit a FOREACH loop before the iterator returns failure. Whenever an iterator fails, it cleans up the stack and restores ESP to the value it had upon encountering the FOREACH statement. However, statements like BREAK, BREAKIF, EXIT, EXITIF, JMP and any other flow of control transfer instructions will not clean

15. In fact, this is exactly how HLA implements iterators and the FOREACH loop. See the volume on Advanced Procedures for more details.

16. Not that it's a good idea to transfer data into or out of any loop using the stack. Such code tends to have lots of errors due to extra pushes or pops appearing in the program.

up the stack if they transfer control out of a FOREACH loop. For example, the following code will leave the activation record for the *range* iterator sitting on the stack:

```
foreach range( 2, 5 ) do

    jmp ExitFor;

endfor;
ExitFor:
```

Depending on the iterator and the code that calls the iterator, prematurely exiting a FOREACH loop without having the iterator return failure and leaving this junk sitting on the stack may have an adverse effect on the operation of your program. Clearly if you've pushed data onto the stack prior to the FOREACH loop, you will not be able to pop that data off unless you manually clean up the stack yourself (this involves saving the value of ESP prior to the FOREACH statement and restoring this value at the *ExitFor* label, above). Also, don't forget that prematurely exiting a FOREACH loop without letting the iterator finish may wind up grabbing some system resources that the iterator would normally free just before returning failure (e.g., calling *free* and closing files).

The volume on Advanced Procedures will go into the details concerning the low-level implementation of iterators. Until then, keep in mind that iterators build their activation records differently than standard procedures. Until you read that chapter, you should not attempt to call an iterator directly (i.e., outside a FOREACH loop) nor should you use the "noframe" option with an iterator. See the chapter on Advanced Procedures for more details on the implementation of iterators.

3.13 Sample Programs

This section presents two sample programs. The first demonstrates the use of iterators using a fibonacci number iterator. The second demonstrates the use of procedural parameters.

3.13.1 Generating the Fibonacci Sequence Using an Iterator

The following program generates the Fibonacci sequence f_1, f_2, f_3, ..., f_{count} where *count* is a parameter. This simple example displays all the fibonacci numbers the iterator generates.

```
program iterDemo;
#include( "stdlib.hhf" )

    //  Basic (recursive version) algorithm for
    //  the fibonacci sequence.
    //
    //  int fib(int N)
    //  {
    //      if(N<=2)
    //          return 1;
    //      else
    //      return fib(N-1) + fib(N-2)
    //  }
    //
    // Iterator (iterative) that computes all the fibonacci
    // numbers between fib(1) and fib(count).

    iterator fib( count:uns32 ); nodisplay;
    var
```

 © 2001, By Randall Hyde

```
        lastVal:        uns32;
        BeforeLastVal:  uns32;

    begin fib;

        if( count > 0 ) then

            mov( 0, BeforeLastVal );
            mov( 1, eax );
            mov( eax, lastVal );

            // Handle fib(1) as a special case.

            yield();
            dec( count );

            // Okay, handle fib(2)..fib(count) here.

            while( @nz ) do

                // Compute fib(n) = fib(n-1) + fib(n-2).
                // and then copy fib(n-1) {lastVal} to
                // fib(n-2) {BeforeLastVal} and store the
                // current result into lastVal so we'll
                // have the n-1 and n-2 values on the next
                // call.

                mov( lastVal, eax );
                add( BeforeLastVal, eax );
                mov( lastVal, BeforeLastVal );
                mov( eax, lastVal );

                // Yield fib(n) to the FOREACH loop.

                yield();

                // Repeat this iterator the specified number
                // of times.

                dec( count );

            endwhile;

        endif;


    end fib;


static
    iteration:uns32;

begin iterDemo;

    // Display the fibonacci sequence for the first
    // ten fibonacci numbers.

    mov( 1, iteration );
    foreach fib( 10 ) do

        stdout.put( "fib(", iteration, ") = ", (type uns32 eax), nl );
```

 © 2001, By Randall Hyde

```
        inc( iteration );

    endfor;

end iterDemo;
```

3.13.2 Outer Product Computation with Procedural Parameters

The following program generates an *addition table*, a *subtraction table*, or a *multiplication table* based on user inputs. These tables are computed using an *outer product* calculation and procedural parameters. An outer product is simply the process of computing all the values for the elements of a matrix by using the row and column indices as inputs to some function (e.g., addition, subtraction, or multiplication).

```
program funcTable;
#include( "stdlib.hhf" )

static
    size: uns32;
    ftbl: array.dArray( uns32, 2 );


    // GenerateTable-
    //
    // This function computes the "Outer Product".  That is,
    // take the cartesian product of the indices into
    // the rows and columns of this array [(0,0), (0,1), ... (0,size-1),
    // (1,0), (1,1), ..., (size-1,size-1)], then feed the left and
    // right values of each coordinate to the "func" procedure passed
    // as a parameter.  Whatever result the function returns, store that
    // into element (l,r) of the ftbl array.

    procedure GenerateTable( func:procedure( l:uns32; r:uns32 )); nodisplay;
    begin GenerateTable;

        push( eax );
        push( ebx );
        push( ecx );
        push( edi );

        for( mov( 0, ebx ); ebx < size; inc( ebx )) do

            for( mov( 0, ecx ); ecx < size; inc( ecx )) do

                array.index( edi, ftbl, ebx, ecx );
                func( ebx, ecx );
                mov( eax, [edi] );

            endfor;

        endfor;

        pop( edi );
        pop( ecx );
        pop( ebx );
        pop( eax );

    end GenerateTable;
```

 © 2001, By Randall Hyde

```
    // The following functions compute the various
    // values used to fill the table (obviously,
    // "+" = addFunc, "-" = subFunc, and "*" = mulFunc).

    procedure addFunc( left:uns32; right:uns32 ); nodisplay;
    begin addFunc;

        mov( left, eax );
        add( right, eax );

    end addFunc;

    procedure subFunc( left:uns32; right:uns32 ); nodisplay;
    begin subFunc;

        mov( left, eax );
        sub( right, eax );

    end subFunc;

    procedure mulFunc( left:uns32; right:uns32 ); nodisplay;
    begin mulFunc;

        mov( left, eax );
        intmul( right, eax );

    end mulFunc;



begin funcTable;

    stdout.put( "Function table generator: " nl );
    stdout.put( "------------------------ " nl nl );

    // Get the size of the function table from the user:

    forever

        try

            stdout.put( "Enter the size of the matrix: " );
            stdin.getu32();
            bound( eax, 1, 20 );
            unprotected break;

          exception( ex.ConversionError )

            stdout.put( "Illegal character, re-enter" nl );

          exception( ex.ValueOutOfRange )

            stdout.put( "Value out of range (1..20), please re-enter" nl );

          exception( ex.BoundInstr )

            stdout.put( "Value out of range (1..20), please re-enter" nl );


        endtry;
```

```
endfor;

// Allocate storage for the function table:

mov( eax, size );
array.daAlloc( ftbl, size, size );


// Get the function from the user:

stdout.put( "What type of table do you want to generate?" nl nl );
stdout.put( "+) Addition" nl );
stdout.put( "-) Subtraction" nl );
stdout.put( "*) Multiplication" nl );
stdout.newln();
repeat

    stdout.put( "Choice? (+, -, *): " );
    stdin.FlushInput();
    stdin.getc();

until( al in {'+', '-', '*'} );

// Fill in the entries in the table:

if( al = '+' ) then

    GenerateTable( &addFunc );

elseif( al = '-' ) then

    GenerateTable( &subFunc );

elseif( al = '*' ) then

    GenerateTable( &mulFunc );

endif;

// Display the column labels across the top:

stdout.put( nl nl "      " );
for( mov( 0, ebx ); ebx < size; inc( ebx )) do

    stdout.put( (type uns32 ebx):5 );

endfor;
stdout.newln();
stdout.put( "      " );
for( mov( 0, ebx ); ebx < size; inc( ebx )) do

    stdout.put( "-----" );

endfor;
stdout.newln();

// Display the row labels and fill in the table.
// Note that this code prints the result as int32
// rather than uns32 because the subFunc function
// returns negative values.
```

 © 2001, By Randall Hyde

```
    for( mov( 0, ebx); ebx < size; inc( ebx )) do

        stdout.put( (type uns32 ebx):4, ": " );
        for( mov( 0, ecx); ecx < size; inc( ecx )) do

            array.index( edi, ftbl, ebx, ecx );
            stdout.puti32size( [edi], 5, ' ' );

        endfor;
        stdout.newln();

    endfor;

end funcTable;
```

3.14 Putting It All Together

In this chapter you saw the low level implementation of procedures and calls to procedures. You learned more about passing parameters by value and reference and you also learned a little more about local variables. This chapter discussed activations records and HLA procedure options. Finally, this chapter wraps up with a discussion of iterators and the FOREACH loop

Your journey through procedures is hardly complete, however. The next volume presents new ways to pass parameters, discusses nested procedures, and explains the low-level implementation of iterators. For more details, see the next volume in this series.

 © 2001, By Randall Hyde

 © 2001, By Randall Hyde