Volume Five: Advanced Procedures

This volume begins the discussion of some specialized information that is of interest to those wanting to become advanced assembly language programmers. Many of the techniques appearing in this volume rarely find their way into typical assembly language programs; this is not because this material is unimportant, but, rather, because most assembly language programs are not written by advanced assembly language programmers. This volume is also of interest to those intending to write compilers or other language translators. This volume discusses the run-time environments that may high level languages use.

Volume Five: Advanced Procedures

Thunks Chapter One

1.1 Chapter Overview

This chapter discusses thunks which are special types of procedures and procedure calls you can use to defer the execution of some procedure call. Although the use of thunks is not commonplace in standard assembly code, their semantics are quite useful in AI (artificial intelligence) and other programs. The proper use of thunks can dramatically improve the performance of certain programs. Perhaps a reason thunks do not find extensive use in assembly code is because most assembly language programmers are unaware of their capabilities. This chapter will solve that problem by presenting the definition of thunks and describe how to use them in your assembly programs.

1.2 First Class Objects

The actual low-level implementation of a thunk, and the invocation of a thunk, is rather simple. However, to understand why you would want to use a thunk in an assembly language program we need to jump to a higher level of abstraction and discuss the concept of *First Class Objects*.

A first class object is one you can treat like a normal scalar data variable. You can pass it as a parameter (using any arbitrary parameter passing mechanism), you can return it as a function result, you can change the object’s value via certain legal operations, you can retrieve its value, and you can assign one instance of a first class object to another. An *int32* variable is a good example of a first class object.

Now consider an array. In many languages, arrays are not first class objects. Oh, you can pass them as parameters and operate on them, but you can’t assign one array to another nor can you return an array as a function result in many languages. In other languages, however, all these operations are permissible on arrays so they are first class objects (in such languages).

A statement sequence (especially one involving procedure calls) is generally not a first class object in many programming languages. For example, in C/C++ or Pascal/Delphi you cannot pass a sequence of statements as a parameter, assign them to a variable, return them as a function result, or otherwise operate on them as though they were data. You cannot create arbitrary arrays of statements nor can you ask a sequence of statements to execute themselves except at their point of declaration.

If you’ve never used a language that allows you to treat executable statements as data, you’re probably wondering why anyone would ever want to do this. There are, however, some very good reasons for wanting to treat statements as data and execute them on demand. If you’re familiar with the C/C++ programming language, consider the C/C++ "?" operator:

```
expr ? Texpr: Fexpr
```

For those who are unfamiliar with the "?" operator, it evaluates the first expression (*expr*) and then returns the value of *Texpr* if *expr* is true, it evaluates and returns *Fexpr* if *expr* evaluates false. Note that this code does not evaluate *Fexpr* if *expr* is true; likewise, it does not evaluate *Texpr* if *expr* is false. Contrast this with the following C/C++ function:

```
int ifexpr( int x, int t, int f )
{
   if( x ) return t;
   return f;
}
```

A function call of the form "ifexpr(expr, Texpr, Fexpr);" is not semantically equivalent to "expr ? Texpr : Fexpr". The *ifexpr* call always evaluates all three parameters while the conditional expression operator ("?") does not. If either *Texpr* or *Fexpr* produces a side-effect, then the call to *ifexpr* may produce a different result than the conditional operator, e.g.,

```
i = (x==y) ? a++ : b--;
j = ifexpr( x==y, c++, d-- );
```

In this example either *a* is incremented or *b* is decremented, but not both because the conditional operator only evaluates one of the two expressions based on the values of *x* and *y*. In the second statement, however, the code both increments *c* and decrements *d* because C/C++ always evaluates all value parameters before calling the function; that is, C/C++ eagerly evaluates function parameter expressions (while the conditional operator uses deferred evaluation).

Supposing that we wanted to defer the execution of the statements "c++" and "d--" until inside the function's body, this presents a classic case where it would be nice to treat a pair of statements as first class objects. Rather than pass the value of "c++" or "d--" to the function, we pass the actual statements and expand these statements inside the function wherever the format parameter occurs. While this is not possible in C/C++, it is possible in certain languages that support the use of statements as first class objects. Naturally, if it can be done in any particular language, then it can be done in assembly language.

Of course, at the machine code level a statement sequence is really nothing more than a sequence of bytes. Therefore, we could treat those statements as data by directly manipulating the object code associated with that statement sequence. Indeed, in some cases this is the best solution. However, in most cases it will prove too cumbersome to manipulate a statement sequence by directly manipulating its object code. A better solution is to use a pointer to the statement sequence and CALL that sequence indirectly whenever we want to execute it. Using a pointer in this manner is usually far more efficient that manipulating the code directly, especially since you rarely change the instruction sequence itself. All you really want to do is defer the execution of that code. Of course, to properly return from such a sequence, the sequence must end with a RET instruction. Consider the following HLA implementation of the "ifexpr" function given earlier:

```
procedure ifexpr( expr:boolean; trueStmts:dword; falseStmts:dword );
    returns( "eax" );

begin ifexpr;

    if( expr ) then

        call( trueStmts );

    else

        call( falseStmts );

    endif;

end ifexpr;
    .
    .
    .
    jmp overStmt1;
stmt1: mov( c, eax );
        inc( c );
        ret();

overStmt1:
        jmp overStmt2
stmt2: mov( d, eax );
        dec( d );
        ret();

overStmt2:
ifexpr( exprVal, &stmt1, &stmt2 );
```

(for reasons you'll see shortly, this code assumes that the *c* and *d* variables are global, static, objects.)

Notice how the code above passes the addresses of the *stmt1* and *stmt2* labels to the *ifexpr* procedure. Also note how the code sequence above jumps over the statement sequences so that the code only executes them in the body of the *ifexpr* procedure.

As you can see, the example above creates two mini-procedures in the main body of the code. Within the *ifexpr* procedure the program calls one of these mini-procedures (*stmt1* or *stmt2*). Unlike standard HLA procedures, these mini-procedures do not set up a proper activation record. There are no parameters, there are no local variables, and the code in these mini-procedures does not execute the standard entry or exit sequence. In fact, the only part of the activation record present in this case is the return address.

Because these mini-procedures do not manipulate EBP's value, EBP is still pointing at the activation record for the *ifexpr* procedure. For this reason, the *c* and *d* variables must be global, static objects; you must not declare them in a VAR section. For if you do, the mini-procedures will attempt to access these objects in *ifexpr's* activation record, not and the caller's activation record. This, of course, would return the wrong value.

Fortunately, there is a way around this problem. HLA provides a special data type, known as a thunk, that eliminates this problem. To learn about thunks, keep reading...

1.3 Thunks

A *thunk* is an object with two components: a pointer containing the address of some code and a pointer to an execution environment (e.g., an activation record). Thunks, therefore, are an eight-byte (64-bit) data type, though (unlike a qword) the two pieces of a thunk are independent. You can declare thunk variables in an HLA program, assign one thunk to another, pass thunks as parameters, return them as function results, and, in general, do just about anything that is possible with a 64-bit data type containing two double word pointers.

To declare a thunk in HLA you use the *thunk* data type, e.g.,

```
static
    myThunk: thunk;
```

Like other 64-bit data types HLA does not provide a mechanism for initializing thunks you declare in a static section. However, you'll soon see that it is easy to initialize a thunk within the body of your procedures.

A *thunk* variable holds two pointers. The first pointer, in the L.O. double word of the thunk, points at some execution environment, that is, an activation record. The second pointer, in the H.O. double word of the thunk, points at the code to execute for the thunk.

To "call" a thunk, you simply apply the "()" suffix to the thunk's name. For example, the following "calls" *myThunk* in the procedure where you've declared *myThunk*:

```
myThunk();
```

Thunks never have parameters, so the parameter list must be empty.

A thunk invocation is a bit more involved than a simple procedure call. First of all, a thunk invocation will modify the value in EBP (the pointer to the current procedure's activation record), so the thunk invocation must begin by preserving EBP's value on the stack. Next, the thunk invocation must load EBP with the address of the thunk's execution environment; that is, the code must load the L.O. double word of the thunk value into EBP. Next, the thunk must call the code at the address specified by the H.O. double word of the thunk variable. Finally, upon returning from this code, the thunk invocation must restore the original activation record pointer from the stack. Here's the exact sequence HLA emits to a statement like "myThunk();":

```
push( (type dword myThunk) );      // Pass execution environment as parm.
call( (type dword myThunk[4]) );   // Call the thunk
```

The body of a thunk, that is, the code at the address found in the H.O. double word of the thunk variable, is not a standard HLA procedure. In particular, the body of a thunk does not execute the standard entry or exit sequences for a standard procedure. The calling code passes the pointer to the execution environment (i.e., an activation record) on the stack.. It is the thunk's responsibility to preserve the current value of EBP and load EBP with this value appearing on the stack. After the thunk loads EBP appropriately, it can execute the statements in the body of the thunk, after which it must restore EBP's original value.

Because a thunk variable contains a pointer to an activation record to use during the execution of the thunk's code, it is perfectly reasonable to access local variables and other local objects in the activation record active when you define the thunk's body. Consider the following code:

```
procedure SomeProc;
var
   c: int32;
   d: int32;
   t: thunk;
begin SomeProc;

   mov( ebp, (type dword t));
   mov( &thunk1, (type dword t[4]));
   jmp OverThunk1;
thunk1:
       push( EBP );           // Preserve old EBP value.
       mov( [esp+8], ebp );   // Get pointer to original thunk environment.
       mov( d, eax );
       add( c, eax );
       pop( ebp );            // Restore caller's environment.
       ret( 4 );              // Remove EBP value passed as parameter.
OverThunk1:
       .
       .
       .
       t();   // Computes the sum of c and d into EAX.
```

This example initializes the *t* variable with the value of *SomeProc's* activation record pointer (EBP) and the address of the code sequence starting at label *thunk1*. At some later point in the code the program invokes the thunk which begins by pushing the pointer to *SomeProc's* activation record. Then the thunk executes the PUSH/MOV/MOV/ADD/POP/RET sequence starting at address *thunk1*. Since this code loads EBP with the address of the activation record containing *c* and *d*, this code sequence properly adds these variables together and leaves their sum in EAX. Perhaps this example is not particularly exciting since the invocation of *t* occurs while EBP is still pointing at *SomeProc's* activation record. However, you'll soon see that this isn't always the case.

1.4 Initializing Thunks

In the previous section you saw how to manually initialize a thunk variable with the environment pointer and the address of an in-line code sequence. While this is a perfectly legitimate way to initialize a thunk variable, HLA provides an easier solution: the THUNK statement.

The HLA THUNK statement uses the following syntax:

```
thunk  thunkVar := #{  code sequence  }#;
```

thunkVar is the name of a thunk variable and *code_sequence* is a sequence of HLA statements (note that the sequence does not need to contain the thunk entry and exit sequences. Specifically, it doesn't need the "push(ebp);" and "mov([esp+8]);" instructions at the beginning of the code, nor does it need to end with the "pop(ebp);" and "ret(4);" instructions. HLA will automatically supply the thunk's entry and exit sequences.

Here's the example from the previous section rewritten to use the THUNK statement:

```
procedure SomeProc;
var
   c: int32;
   d: int32;
```

```
    t: thunk;
begin SomeProc;


    thunk  t :=
            #{
                mov( d, eax );
                add( c, eax );
            }#;
        .
        .
        .
        t();   // Computes the sum of c and d into EAX.
```

Note how much clearer and easier to read this code sequence becomes when using the THUNK statement. You don't have to stick in statements to initialize *t*, you don't have to jump over the thunk body, you don't have to include the thunk entry/exit sequences, and you don't wind up with a bunch of statement labels in the code. Of course, HLA emits the same code sequence as found in the previous section, but this form is much easier to read and work with.

1.5 Manipulating Thunks

Since a thunk is a 64-bit variable, you can do anything with a thunk that you can do, in general, with any other qword data object. You can assign one thunk to another, compare thunks, pass thunks a parameters, return thunks as function results, and so on. That is to say, thunks are first class objects. Since a thunk is a representation of a sequence of statements, those statements are effectively first class objects. In this section we'll explore the various ways we can manipulate thunks and the statements associated with them.

1.5.1 Assigning Thunks

To assign one thunk to another you simply move the two double words from the source thunk to the destination thunk. After the assignment, both thunks specify the same sequence of statements and the same execution environment; that is, the thunks are now aliases of one another. The order of assignment (H.O. double word first or L.O. double word first) is irrelevant as long as you assign both double words before using the thunk's value. By convention, most programmers assign the L.O. double word first. Here's an example of a thunk assignment:

```
        mov( (type dword srcThunk), eax );
        mov( eax, (type dword destThunk));
        mov( (type dword srcThunk[4]), eax );
        mov( eax, (type dword destThunk[4]));
```

If you find yourself assigning one thunk to another on a regular basis, you might consider using a macro to accomplish this task:

```
#macro movThunk( src, dest );

        mov( (type dword src), eax );
        mov( eax, (type dword dest));
        mov( (type dword src[4]), eax );
        mov( eax, (type dword dest[4]));

#endmacro;
```

If the fact that this macro's side effect of disturbing the value in EAX is a concern to you, you can always copy the data using a PUSH/POP sequence (e.g., the HLA extended syntax MOV instruction):

```
#macro movThunk( src, dest );

       mov( (type dword src), (type dword dest));
       mov( (type dword src[4]), (type dword dest[4]));

#endmacro;
```

If you don't plan on executing any floating point code in the near future, or you're already using the MMX instruction set, you can also use the MMX MOVQ instruction to copy these 64 bits with only two instructions:

```
       movq( src, mm0 );
       movq( mm0, dest );
```

Don't forget, however, to execute the EMMS instruction before calling any routines that might use the FPU after this sequence.

1.5.2 Comparing Thunks

You can compare two thunks for equality or inequality using the standard 64-bit comparisons (see "Extended Precision Comparisons" on page 857). If two thunks are equal then they refer to the same code sequence with the same execution environment; if they are not equal, then they could have different code sequences or different execution environments (or both) associated with them. Note that it doesn't make any sense to compare one thunk against another for less than or greater than. They're either equal or not equal.

Of course, it's quite easy to have two thunks with the same environment pointer and different code pointers. This occurs when you initialize two thunk variables with separate code sequences in the same procedure, e.g.,

```
   thunk  t1 :=
          #{
             mov( 0, eax );
             mov( i, ebx );
          }#;

   thunk  t2 :=
          #{
             mov( 4, eax );
             mov( j, ebx );
          }#;

// At this point, t1 and t2 will have the same environment pointer
// (EBP's value) but they will have different code pointers.
```

Note that it is quite possible for two thunks to refer to the same statement sequence yet have different execution environments. This can occur when you have a recursive function that initializes a pair of thunk variables with the same instruction sequence on different recursive calls of the function. Since each recursive invocation of the function will have its own activation record, the environment pointers for the two thunks will be different even though the pointers to the code sequence are the same. However, if the code that initializes a specific thunk is not recursive, you can sometimes compare two thunks by simply comparing their code pointers (the H.O. double words of the thunks) if you're careful about never using thunks once their execution environment goes away (i.e., the procedure in which you originally assigned the thunk value returns to its caller).

1.5.3 Passing Thunks as Parameters

Since the thunk data type is effectively equivalent to a qword type, there is little you can do with a qword object that you can't also do with a thunk object. In particular, since you can pass qwords as parameters you can certainly pass thunks as parameters to procedures.

To pass a thunk by value to a procedure is very easy, simply declare a formal parameter using the thunk data type:

```
procedure HasThunkParm( t:thunk );
var
    i:integer;
begin HasThunkParm;

    mov( 1, i );
    t();            // Invoke the thunk passed as a parameter.
    mov( i, eax );  // Note that t does not affect our environment.

end HasThunkParm;
        .
        .
        .
    thunk  thunkParm :=
           #{
               mov( 0, i );  // Not the same "i" as in HasThunkParm!
           }#;

    HasThunkParm( thunkParm );
```

Although a thunk is a pointer (a pair of pointers, actually), you can still pass thunks by value. Passing a thunk by value passes the values of those two pointer objects to the procedure. The fact that these values are the addresses of something else is not relevant, you're passing the data by value.

HLA automatically pushes the value of a thunk on the stack when passing a thunk by value. Since thunks are 64-bit objects, you can only pass them on the stack, you cannot pass them in a register[1]. When HLA passes a thunk, it pushes the H.O. double word (the code pointer) of the thunk first followed by the L.O. double word (the environment pointer). This way, the two pointers are situated on the stack in the same order they normally appear in memory (the environment pointer at the lowest address and the code pointer at the highest address).

If you decide to manually pass a thunk on the stack yourself, you must push the two halves of the thunk on the stack in the same order as HLA, i.e., you must push the H.O. double word first and the L.O. double word second. Here's the call to *HasThunkParm* using manual parameter passing:

```
    push( (type dword thunkParm[4]) );
    push( (type dword thunkParm) );
    call HasThunkParm;
```

You can also pass thunks by reference to a procedure using the standard pass by reference syntax. Here's a typical procedure prototype with a pass by reference thunk parameter:

```
procedure refThunkParm( var t:thunk ); forward;
```

When you pass a thunk by reference, you're passing a pointer to the thunk itself, not the pointers to the thunk's execution environment or code sequence. To invoke such a thunk you must manually dereference the pointer to the thunk, push the pointer to the thunk's execution environment, and indirectly call the code sequence. Here's an example implementation of the *refThunkParm* prototype above:

1. Technically, you could pass a thunk in two 32-bit registers. However, you will have to do this manually; HLA will not automatically move the two pointers into two separate registers for you.

```
procedure refThunkParm( var t:thunk );
begin refThunkParm;

    push( eax );
        .
        .
        .
    mov( t, eax );                 // Get pointer to thunk object.
    push( [eax] );                 // Push pointer to thunk's environment.
    call( (type dword [eax+4]) );  // Call the code sequence.
        .
        .
        .
    pop( eax );

end refThunkParm;
```

Of course, one of the main reasons for passing an object by reference is so you can assign a value to the actual parameter value. Passing a thunk by reference provides this same capability – you can assign a new code sequence address and execution environment pointer to a thunk when you pass it by reference. However, always be careful when assigning values to thunk reference parameters within a procedure that you specify an execution environment that will still be valid when the code actually invokes the thunk. We'll explore this very problem in a later section of this chapter (see "Activation Record Lifetimes and Thunks" on page 1288).

Although we haven't yet covered this, HLA does support several other parameter passing mechanisms beyond pass by value and pass by reference. You can certainly pass thunks using these other mechanisms. Indeed, thunks are the basis for two of HLA's parameter passing mechanisms: pass by name and pass by evaluation. However, this is getting a little ahead of ourselves; we'll return to this subject in a later chapter in this volume.

1.5.4 Returning Thunks as Function Results

Like any other first class data object, we can also return thunks as the result of some function. The only complication is the fact that a thunk is a 64-bit object and we normally return function results in a register. To return a full thunk as a function result, we're going to need to use two registers or a memory location to hold the result.

To return a 64-bit (non-floating point) value from a function there are about three or four different locations where we can return the value: in a register pair, in an MMX register, on the stack, or in a memory location. We'll immediately discount the use of the MMX registers since their use is not general (i.e., you can't use them simultaneously with floating point operations). A global memory location is another possible location for a function return result, but the use of global variables has some well-known deficiencies, especially in a multi-threaded/multi-tasking environment. Therefore, we'll avoid this solution as well. That leaves using a register pair or using the stack to return a thunk as a function result. Both of these schemes have their advantages and disadvantages, we'll discuss these two schemes in this section.

Returning thunk function results in registers is probably the most convenient way to return the function result. The big drawback is obvious – it takes two registers to return a 64-bit thunk value. By convention, most programmers return 64-bit values in the EDX:EAX register pair. Since this convention is very popular, we will adopt it in this section. Keep in mind, however, that you may use almost any register pair you like to return this 64-bit value (though ESP and EBP are probably off limits).

When using EDX:EAX, EAX should contain the pointer to the execution environment and EDX should contain the pointer to the code sequence. Upon return from the function, you should store these two registers into an appropriate thunk variable for future use.

To return a thunk on the stack, you must make room on the stack for the 64-bit thunk value prior to pushing any of the function's parameters onto the stack. Then, just before the function returns, you store the thunk result into these locations on the stack. When the function returns it cleans up the parameters it pushed on the stack but it does not free up the thunk object. This leaves the 64-bit thunk value sitting on the top of the stack after the function returns.

The following code manually creates and destroys the function's activation record so that it can specify the thunk result as the first two parameters of the function's parameter list:

```
procedure RtnThunkResult
(
    ThunkCode:dword;   // H.O. dword of return result goes here.
    ThunkEnv:dword;    // L.O. dword of return result goes here.
    selection:boolean; // First actual parameter.
    tTrue: thunk;      // Return this thunk if selection is true.
    tFalse:thunk       // Return this thunk if selection is false.
); @nodisplay; @noframe;
begin RtnThunkResult;

    push( ebp );       // Set up the activation record.
    mov( esp, ebp );
    push( eax );

    if( selection ) then

        mov( (type dword tTrue), eax );
        mov( eax, ThunkEnv );
        mov( (type dword tTrue[4]), eax );
        mov( eax, ThunkCode );

    else

        mov( (type dword tFalse), eax );
        mov( eax, ThunkEnv );
        mov( (type dword tFalse[4]), eax );
        mov( eax, ThunkCode );

    endif;

    // Clean up the activation record, but leave the eight
    // bytes of the thunk return result on the stack when this
    // function returns.

    pop( eax );
    pop( ebp );
    ret( _parms_ - 8 );  // _parms_ is total # of bytes of parameters (28).

end RtnThunkResult;
        .
        .
        .
// Example of call to RtnThunkResult and storage of return result.
// (Note passing zeros as the thunk values to reserve storage for the
// thunk return result on the stack):

    RtnThunkResult( 0, 0, ChooseOne, t1, t2 );
    pop( (type dword SomeThunkVar) );
    pop( (type dword SomeThunkVar[4]) );
```

If you prefer not to list the thunk parameter as a couple of pseudo-parameters in the function's parameter list, you can always manually allocate storage for the parameters prior to the call and refer to them using the "[ESP+disp]" or "[EBP+disp]" addressing mode within the function's body.

1.6 Activation Record Lifetimes and Thunks

There is a problem that can occur when using thunks in your applications: it's quite possible to invoke a thunk long after the associated execution environment (activation record) is no longer valid. Consider the following HLA code that demonstrates this problem:

```
static
    BadThunk: thunk;

    procedure proc1;
    var
        i:int32;
    begin proc1;

        thunk  BadThunk :=
               #{
                   stdout.put( "i = ", i, nl );
               #};
        mov( 25, i );

    end proc1;

    procedure proc2;
    var
        j:int32;
    begin proc2;

        mov( 123, j );
        BadThunk();

    end proc2;
        .
        .
        .
```

If the main program in this code fragment calls *proc1* and then immediately calls *proc2*, this code will probably print "i = 123" although there is no guarantee this will happen (the actual result depends on a couple of factors, although "i = 123" is the most likely output).

The problem with this code example is that *proc1* initializes *BadThunk* with the address of an execution environment that is no longer "live" when the program actually executes the thunk's code. The *proc1* procedure constructs its own activation record and initializes the variable *i* in this activation record with the value 25. This procedure also initializes *BadThunk* with the address of the code sequence containing the *stdout.put* statement and it initializes *BadThunk's* execution environment pointer with the address of *proc1's* activation record. Then *proc1* returns to its caller. Unfortunately upon returning to its caller, *proc1* also obliterates its activation record even though *BadThunk* still contains a pointer into this area of memory. Later, when the main program calls *proc2*, *proc2* builds its own activation record (most likely over the top of *proc1's* old activation record). When *proc2* invokes *BadThunk*, *BadThunk* uses the original pointer to *proc1's* activation record (which is now invalid and probably points at *proc2's* activation record) from which to fetch *i's* value. If nothing extra was pushed or popped between the *proc1* invocation and the *proc2* invocation, then *j's* value in *proc2* is probably at the same memory location as *i* was in *proc1's* invocation. Hence, the *stdout.put* statement in the thunk's code will print *j's* value.

This rather trivial example demonstrates an important point about using thunks – you must always ensure that a thunk's execution environment is still valid whenever you invoke a thunk. In particular, if you use HLA's THUNK statement to automatically initialize a thunk variable with the address of a code sequence and the execution environment of the current procedure, you must not invoke that thunk once the procedure returns to its caller; for at that point the thunk's execution environment pointer will not be valid.

This discussion does not suggest that you can only use a thunk within the procedure in which you assign a value to that thunk. You may continue to invoke a thunk, even from outside the procedure whose activation record the thunk references, until that procedure returns to its caller. So if that procedure calls some other procedure (or even itself, recursively) then it is legal to call the thunk associated with that procedure.

1.7 Comparing Thunks and Objects

Thunks are very similar to objects insofar as you can easily implement an abstract data type with a thunk. Remember, an abstract data type is a piece of data and the operation(s) on that data. In the case of a thunk, the execution environment pointer can point at the data while the code address can point at the code that operates on the data. Since you can also use objects to implement abstract data types, one might wonder how objects and thunks compare to one another.

Thunks are somewhat less "structured" than objects. An object contains a set of data values and operations (methods) on those data values. You cannot change the data values an object operates upon without fundamentally changing the object (i.e., selecting a different object in memory). It is possible, however, to change the execution environment pointer in a thunk and have that thunk operate on fundamentally different data. Although such a course of action is fraught with difficulty and very error-prone, there are some times when changing the execution environment pointer of a thunk will produce some interesting results. This text will leave it up to you to discover how you could abuse thunks in this fashion.

1.8 An Example of a Thunk Using the Fibonacci Function

By now, you're probably thinking "thunks may be interesting, but what good are they?" The code associated with creating and invoking thunks is not spectacularly efficient (compared, say, to a straight procedure call). Surely using thunks must negatively impact the execution time of your code, eh? Well, like so many other programming constructs, the misuse and abuse of thunks can have a negative impact on the execution time of your programs. However, in an appropriate situation thunks can dramatically improve the performance of your programs. In this section we'll explore one situation where the use of thunks produces an amazing performance boost: the calculation of a Fibonacci number.

Earlier in this text there was an example of a Fibonacci number generation program (see "Fibonacci Number Generation" on page 50). As you may recall, the Fibonacci function fib(n) is defined recursively for n>= 1 as follows:

```
fib(1) = 1;
fib(2) = 1;
fib( n ) = fib( n-1 ) + fib( n-2 )
```

One problem with this recursive definition for *fib* is that it is extremely inefficient to compute. The number of clock cycles this particular implementation requires to execute is some exponential factor of *n*. Effectively, as *n* increases by one this algorithm takes twice as long to execute.

The big problem with this recursive definition is that it computes the same values over and over again. Consider the statement "fib(n) = fib(n-1) + fib(n-2)". Note that the computation of fib(n-1) also computes fib(n-2) (since fib(n-1) = fib(n-2) + fib(n-3) for all n >=4). Although the computation of fib(n-1) computes the value of fib(n-2) as part of its calculation, this simple recursive definition doesn't save that result, so it must recompute fib(n-2) upon returning from fib(n-1) in order to complete the calculation of fib(n).

Since the calculation of Fib(n-1) generally computes Fib(n-2) as well, what would be nice is to have this function return both results simultaneously; that is, not only should Fib(n-1) return the Fibonacci number for n-1, it should also return the Fibonacci number for n-2 as well. In this example, we will use a thunk to store the result of Fib(n-2) into a local variable in the Fibonacci function.

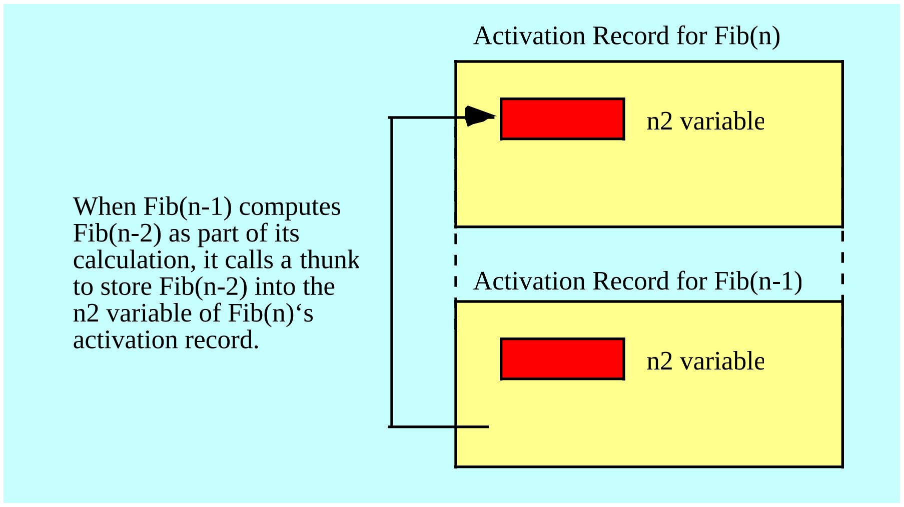
Activation Record for Fib(n)
n2 variable
When Fib(n-1) computes Fib(n-2) as part of its calculation, it calls a thunk to store Fib(n-2) into the n2 variable of Fib(n)'s activation record.
Activation Record for Fib(n-1)
n2 variable

Figure 1.1 Using a Thunk to Set the Fib(n-2) Value in a Different Activation Record

The following program provides two versions of the Fibonacci function: one that uses thunks to pass the Fib(n-2) value back to a previous invocation of the function. Another version of this function computes the Fibonacci number using the traditional recursive definition. The program computes the running time of both implementations and displays the results. Without further ado, here's the code:

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

// Fibonacci function using a thunk to calculate fib(n-2)
// without making a recursive call.

procedure fib( n:uns32; nm2:thunk ); nodisplay; returns( "eax" );
var
    n2: uns32;      // A recursive call to fib stores fib(n-2) here.
    t:  thunk;      // This thunk actually stores fib(n-2) in n2.

begin fib;

    // Special case for n = 1, 2.  Just return 1 as the
    // function result and store 1 into the fib(n-2) result.

    if( n <= 2 ) then

        mov( 1, eax );  // Return as n-1 value.
        nm2();          // Store into caller as n-2 value.

    else

        // Create a thunk that will store the fib(n-2) value
        // into our local n2 variable.

        thunk   t :=
                #{
                    mov( eax, n2 );
                }#;
```

```
            mov( n, eax );
            dec( eax );
            fib( eax, t );  // Compute fib(n-1).

            // Pass back fib(n-1) as the fib(n-2) value to a previous caller.

            nm2();


            // Compute fib(n) = fib(n-1) [in eax] + fib(n-2) [in n2]:

            add( n2, eax );

        endif;

    end fib;


    // Standard fibonacci function using the slow recursive implementation.

    procedure slowfib( n:uns32 ); nodisplay; returns( "eax" );
    begin slowfib;

        // For n= 1,2 just return 1.

        if( n <= 2 ) then

            mov( 1, eax );

        else

            // Return slowfib(n-1) + slowfib(n-2) as the function result:

            dec( n );
            slowfib( n );   // compute fib(n-1)
            push( eax );    // Save fib(n-1);

            dec( n );       // compute fib(n-2);
            slowfib( n );

            add( [esp], eax );  // Compute fib(n-1) [on stack] + fib(n-2) [in eax].
            add( 4, esp );      // Remove old value from stack.

        endif;

    end slowfib;


    var
        prevTime:dword[2];      // Used to hold 64-bit result from RDTSC instr.
        qw: qword;              // Used to compute difference in timing.
        dummy:thunk;            // Used in original calls to fib.

    begin fibThunk;

        // "Do nothing" thunk used by the initial call to fib.
        // This thunk simply returns to its caller without doing
        // anything.

        thunk dummy := #{ }#;
        // Call the fibonacci routines to "prime" the cache:

        fib( 1, dummy );
        slowfib( 1 );

        // Okay, compute the times for the two fibonacci routines for
        // values of n from 1 to 32:

        for( mov( 1, ebx ); ebx < 32; inc( ebx )) do

            // Read the time stamp counter before calling fib:
            rdtsc();
            mov( eax, prevTime );
            mov( edx, prevTime[4] );

            fib( ebx, dummy );
            mov( eax, ecx );

            // Read the timestamp counter and compute the approximate running
            // time of the current call to fib:

            rdtsc();
            sub( prevTime, eax );
            sbb( prevTime[4], edx );
            mov( eax, (type dword qw));
            mov( edx, (type dword qw[4]));

            // Display the results and timing from the call to fib:

            stdout.put
            (
                "n=",
                (type uns32 ebx):2,
                " fib(n) = ",
                (type uns32 ecx):7,
                " time="
            );
            stdout.putu64size( qw, 5, ' ' );


            // Okay, repeat the above for the slowfib implementation:

            rdtsc();
            mov( eax, prevTime );
            mov( edx, prevTime[4] );

            slowfib( ebx );
            mov( eax, ecx );
            rdtsc();
            sub( prevTime, eax );
            sbb( prevTime[4], edx );
            mov( eax, (type dword qw));
            mov( edx, (type dword qw[4]));

            stdout.put( " slowfib(n) = ", (type uns32 ecx ):7, " time = " );
            stdout.putu64size( qw, 8, ' ' );
            stdout.newln();

        endfor;
```

```
end fibThunk;
```

Program 1.1 Fibonacci Number Generation Using Thunks

This (relatively) simple modification to the Fibonacci function produces a dramatic difference in the run-time of the code. The run-time of the thunk implementation is now well within reason. The following table lists the same run-times of the two functions (thunk implementation vs. standard recursive implementation). As you can see, this small change to the program has made a very significant difference.

Table 1: Running Time of the FIB and SlowFib Functions

n	Fib Execution Time (Thunk Implementation)[a]	SlowFib Execution Time (Recursive Implementation)
1	60	97
2	152	100
3	226	166
4	270	197
5	302	286
6	334	414
7	369	594
8	397	948
9	432	1513
10	473	2421
11	431	3719
12	430	6010
13	467	9763
14	494	15758
15	535	25522
16	564	41288
17	614	66822
18	660	108099
19	745	174920

Table 1: Running Time of the FIB and SlowFib Functions

n	Fib Execution Time (Thunk Implementation)[a]	SlowFib Execution Time (Recursive Implementation)
20	735	283001
21	791	457918
22	886	740894
23	943	1198802
24	919	1941077
25	966	3138466
26	1015	5094734
27	1094	8217396
28	1101	13297000
29	1158	21592819
30	3576[b]	34927400
31	1315	56370705

a. All times are in CPU cycles as measured via RDTSC on a Pentium II processor.

b. This value was not recorded properly because of OS overhead.

Note that a thunk implementation of the Fibonacci function is not the only way to improve the performance of this function. One could have just as easily (more easily, in fact) passed the address of the local *n2* variable by reference and had the recursive call store the Fib(n-2) value directly into the *n2* variable. For that matter, one could have written an interactive (rather than recursive) solution to this problem that computes the Fibonacci number very efficiently. However, alternate solutions to Fibonacci aside, this example does clearly demonstrate that the use of a thunk in certain situations can dramatically improve the performance of an algorithm.

1.9 Thunks and Artificial Intelligence Code

Although the use of thunks to compute the Fibonacci number in the previous section produced a dramatic performance boost, thunks clearly were not necessary for this operation (indeed, not too many people really need to compute Fibonacci numbers, for that matter). Although this example demonstrates that thunks can improve performance in various situations, it does not demonstrate the need for thunks. After all, there are even more efficient implementations of the Fibonacci function (though nothing quite so dramatic as the difference between Fib and SlowFib, which went from exponential to linear execution time) that do not involve the use of thunks. So although the use of thunks can increase the performance of the Fibonacci function (over the execution time of the standard recursive implementation), the example in the previous section does not demonstrate the need for thunks since there are better implementations of the Fibonacci function that do not use thunks. In this section we will explore some types of calculations for which thunks present a very good, if not the best, solution.

In the field of Artificial Intelligence (AI from this point forward) researchers commonly use interpreted programming languages such as LISP or Prolog to implement various algorithms. Although you could write an AI program in just about any language, these interpreted languages like LISP and Prolog have a couple of benefits that help make writing AI programs much less difficult. Among a long list of other features, two important features stand out: (1) statements in these languages are first class objects, (2) these languages can defer the evaluation of function parameters until the parameters' values are actually needed (lazy evaluation). Since thunks provide exactly these two features in an HLA program, it should come as no surprise that you can use thunks to implement various AI algorithm in assembly language.

Before going too much farther, you should realize that AI programs usually take advantage of many other features in LISP and Prolog besides the two noted above. Automatic dynamic memory allocation and garbage collection are two big features that many AI programs use, for example. Also, the run-time interpretation of language statements is another big feature (i.e., the user of a program can input a string containing a LISP or Prolog statement and the program can execute this string as part of the program). Although it is certainly possible to achieve all this in an assembly language program, such support built into languages like LISP and Prolog may require (lots of) additional coding in assembly language. So please don't allow this section to trivialize the effort needed to write AI programs in assembly language; writing (good) AI programs is difficult and tools like LISP and Prolog can reduce that effort.

Of course, a major problem with languages like LISP and Prolog is that programs written in these (interpreted) languages tend to run very slow. Some people may argue that there isn't a sufficient difference in performance between programs written in C/C++ and assembly to warrant the extra effort of writing the code in assembly; such a claim is difficult to make about LISP or Prolog programs versus an assembly equivalent. Whereas a well-written assembly program may be only a couple of times faster than a well-written C/C++ program, a well-written assembly program will probably be tens, if not hundreds or thousands, of times faster than the equivalent LISP or Prolog code. Therefore, reworking at least a portion of an AI program written in one of these interpreted languages can produce a very big boost in the performance of the AI application.

Traditionally, one of the big problem areas AI programmers have had when translating their applications to a lower-level language has been the issue of "function calls (statements) as first class objects and the need for lazy evaluation." Traditional third generation programming languages like C/C++ and Pascal simply do not provide these facilities. AI applications that make use of these facilities in languages like LISP or Prolog often have to be rewritten in order to avoid the use of these features in the lower-level languages. Assembly language doesn't suffer from this problem. Oh, it may be difficult to implement some feature in assembly language, but if it can be done, it can be done in assembly language. So you'll never run into the problem of assembly language being unable to implement some feature from LISP, Prolog, or some other language.

Thunks are a great vehicle for deferring the execution of some code until a later time (or, possibly, forever). One application area where deferred execution is invaluable is in a game. Consider a situation in which the current state of a game suggests one of several possible moves a piece could make based on a decision made by an adversary (e.g., a particular chess piece could make one of several different moves depending on future moves by the other color). You could represent these moves as a list of thunks and executing the move by selecting and executing one of the thunks from the list at some future time. You could base the selection of the actual thunk to execute on adversarial moves that occur at some later time.

Thunks are also useful for passing results of various calculations back to several different points in your code. For example, in a multi-player strategy game the activities of one player could be broadcast to a list of interested players by having those other players "register" a thunk with the player. Then, whenever the player does something of interest to those other players, the program could execute the list of thunks and pass whatever data is important to those other players via the thunks.

1.10 Thunks as Triggers

Thunks are also useful as *triggers*. A trigger is a device that fires whenever a certain condition is met. Probably the most common example of a trigger is a database trigger that executes some code whenever some condition occurs in the database (e.g., the entry of an invalid data or the update of some field). Triggers are useful insofar as they allow the use of *declarative programming* in your assembly code. Declarative programming consists of some declarations that automatically execute when a certain condition exists. Such declarations do not execute sequentially or in response to some sort of call. They are simply part of the programming environment and automatically execute whenever appropriate. At the machine level, of course, some sort of call or jump must be made to such a code sequence, but at a higher level of abstraction the code seems to "fire" (execute) all on its own.

To implement declarative programming using triggers you must get in the habit of writing code that always calls a "trigger routine" (i.e., a thunk) at any given point in the code where you would want to handle some event. By default, the trigger code would be an empty thunk, e.g.:

```
procedure DefaultTriggerProc; @nodisplay; @noframe;
begin DefaultTriggerProc;

    // Immediately return to the caller and pop off the environment
    // pointer passed to us (probably a NULL pointer).

    ret(4);

end DefaultTriggerProc;

static
    DefaultTrigger:   thunk: @nostorage;
                      dword 0, &DefaultTriggerProc;
```

The code above consists of two parts: a procedure that corresponds to the default thunk code to execute and a declaration of a default trigger thunk object. The procedure body consists of nothing more than a return that pops the EBP value the thunk invocation pushes and then returns back to the thunk invocation. The default trigger thunk variable contains NULL (zero) for the EBP value and the address of the *DefaultTriggerProc* code as the code pointer. Note that the value we pass as the environment pointer (EBP value) is irrelevant since *DefaultTriggerProc* ignores this value.

To use a trigger, you simply declare a thunk variable like *DefaultTrigger* above and initialize it with the address of the *DefaultTriggerProc* procedure. You will need a separate thunk variable for each trigger event you wish to process; however, you will only need one default trigger procedure (you can initialize all trigger thunks with the address of this same procedure). Generally, these trigger thunks will be global variables so you can access the thunk values throughout your program. Yes, using global variables is often a no-no from a structured point of view, but triggers tend to be global objects that several different procedures share, so using global objects is appropriate here. If using global variables for these thunks offends you, then bury them in a class and provide appropriate accessor methods to these thunks.

Once you have a thunk you want to use as a trigger, you invoke that thunk from the appropriate point in your code. As a concrete example, suppose you have a database function that updates a record in the database. It is common (in database programs) to trigger an event after the update and, possibly, before the update. Therefore, a typical database update procedure might invoke two thunks – one before and one after the body of the update procedure’s code. The following code fragment demonstrates how you code do this:

```
static
    preUpdate:        thunk: @nostorage;
                      dword 0, &DefaultTriggerProc;

    postUpdate:       thunk: @nostorage;
                      dword 0, &DefaultTriggerProc;

        .
        .
        .
procedure databaseUpdate( << appropriate parameters >> );
  << declarations >>
begin databaseUpdate;

    preUpdate();   // Trigger the pre-update event.

    << Body of update procedure >>

    postUpdate();  // Trigger the post-update event.

end databaseUpdate;
```

As written, of course, these triggers don't do much. They call the default trigger procedure that immediately returns. Thus far, the triggers are really nothing more than a waste of time and space in the program. However, since the *preUpdate* and *postUpdate* thunks are variables, we can change their values under program control and redirect the trigger events to different code.

When changing a trigger's value, it's usually a good idea to first preserve the existing thunk data. There isn't any guarantee that the thunk points at the default trigger procedure. Therefore, you should save the value so you can restore it when you're done handling the trigger event (assuming you are writing an event handler that shuts down before the program terminates). If you're setting and restoring a trigger value in a procedure, you can copy the global thunk's value into a local variable prior to setting the thunk and you can restore the thunk from this local variable prior to returning from the procedure:

```
procedure DemoRestoreTrigger;
var
    RestoreTrigger: dword[2];
begin DemoRestoreTrigger;

    // The following three statements "register" a thunk as the
    // "GlobalEvent" trigger:

    mov( (type dword GlobalEvent[0]), RestoreTrigger[0] );
    mov( (type dword GlobalEvent[4]), RestoreTrigger[4] );
    thunk GlobalEvent := #{  <<thunk body >> }#;

    << Body of DemoRestoreTrigger procedure >>

    // Restore the original thunk as the trigger event:

    mov( RestoreTrigger[0], (type dword GlobalEvent[0]) );
    mov( RestoreTrigger[4], (type dword GlobalEvent[4]) );

end DemoRestoreTrigger;
```

Note that this code works properly even if *DemoRestoreTrigger* is recursive since the *RestoreTrigger* variable is an automatic variable. You should always use automatic (VAR) objects to hold the saved values since static objects have only a single instance (which would fail in a multi-threaded environment or if *DemoRestoreTrigger* is recursive).

One problem with the code in the example above is that it replaces the current trigger with a new trigger. While this is sometimes desirable, more often you'll probably want to *chain* the trigger events. That is, rather than having a trigger call the most recent thunk, which returns to the original code upon completion, you'll probably want to call the original thunk you replaced before or after the current thunk executes. This way, if several procedures register a trigger on the same global event, they will all "fire" when the event occurs. The following code fragment shows the minor modifications to the code fragment above needed to pull this off:

```
procedure DemoRestoreTrigger;
```

```
var
    PrevTrigger: thunk;
begin DemoRestoreTrigger;

    // The following three statements "register" a thunk as the
    // "GlobalEvent" trigger:

    mov( (type dword GlobalEvent[0]), (type dword PrevTrigger[0]) );
    mov( (type dword GlobalEvent[4]), (type dword PrevTrigger[4]) );
    thunk GlobalEvent :=
        #{
            PrevThunk();
            <<thunk body >>
        }#;

    << Body of DemoRestoreTrigger procedure >>

    // Restore the original thunk as the trigger event:

    mov( (type dword PrevTrigger[0]), (type dword GlobalEvent[0]) );
    mov( (type dword PrevTrigger[4]), (type dword GlobalEvent[4]) );

end DemoRestoreTrigger;
```

The principal differences between this version and the last is that *PrevTrigger* (a thunk) replaces the *RestoreTrigger* (two double words) variable and the thunk code invokes *PrevTrigger* before executing its own code. This means that the thunk's body will execute *after* all the previous thunks in the chain. If you would prefer to execute the thunks body before all the previous thunks in the chain, then simply invoke the thunk *after* the thunk's body, e.g.,

```
    thunk GlobalEvent :=
        #{
            <<thunk body >>
            PrevThunk();
        }#;
```

In practice, most programs set a trigger event once and let a single, global, trigger handler process events from that point forward. However, if you're writing more sophisticated code that enables and disables trigger events throughout, you might want to write a macro that helps automate saving, setting, and restore thunk objects. Consider the following HLA macro:

```
#macro EventHandler( Event, LocalThunk );

        mov( (type dword Event[0]), (type dword LocalThunk[0]) );
        mov( (type dword Event[4]), (type dword LocalThunk[4]) );
        thunk Event :=

    #terminator EndEventHandler;

        mov( (type dword LocalThunk[0]), (type dword Event[0]) );
        mov( (type dword LocalThunk[4]), (type dword Event[4]) );

#endmacro;
```

This macro lets you write code like the following:

```
procedure DemoRestoreTrigger;
var
    PrevTrigger: thunk;
begin DemoRestoreTrigger;
```

```
    EventHandler( GlobalEvent, PrevTrigger )  // Note: no semicolon here!

        #{
            PrevThunk();
            <<thunk body >>
        }#;

        << Body of DemoRestoreTrigger procedure >>

    EndEventHandler;

end DemoRestoreTrigger;
```

Especially note the comment stating that no semicolon follows the *EventHandler* macro invocation. If you study the *EventHandler* macro carefully, you'll notice that macro ends with the first half of a THUNK statement. The body of the *EventHandler..EndEventHandler* statement (macro invocation) must begin with a thunk body declaration that completes the THUNK statement begun in *EventHandler.* If you put a semicolon at the end of the *EventHandler* statement, this will insert the semicolon into the middle of the THUNK statement, resulting in a syntax error. If these syntactical gymnastics bother you, you can always remove the THUNK statement from the macro and require the end user to type the full THUNK statement at the beginning of the macro body. However, saving this extra type is what macros are all about; most users would probably rather deal with remembering not to put a semicolon at the end of the *EventHandler* statement rather than do this extra typing.

1.11 Jumping Out of a Thunk

Because a thunk is a procedure nested within another procedure's body, there are some interesting situations that can arise during program execution. One such situation is jumping out of a thunk and into the surrounding code during the execution of that thunk. Although it is possible to do this, you must exercise great caution when doing so. This section will discuss the precautions you must take when leaving a thunk other than via a RET instruction.

Perhaps the best place to start is with a couple of examples that demonstrate various ways to abnormally exit a thunk. The first thunk in the example below demonstrates a simple JMP instruction while the second thunk in this example demonstrates leaving a thunk via a BREAK statement.

```
procedure ExitThunks;
var
    jmpFrom:thunk;
    breakFrom:thunk;
begin ExitThunks;

    thunk jmpFrom :=
        #{
            // Just jump out of this thunk and back into
            // the ExitThunks procedure:

            jmp XTlabel;
        }#;

    // Execute the thunk above (which winds up jumping to the
    // XTlabel label below:

    jmpFrom();

    XTlabel:
```

```
    // Create a loop inside the ExitThunks procedure and
    // define a thunk within this loop.  Use a BREAK statement
    // within the thunk to exit the thunk (and loop).

    forever

        thunk breakFrom :=
            #{
                // Break out of this thunk and the surrounding
                // loop via the following BREAK statement:

                break;
            }#;

        // Invoke the thunk (which causes use to exit from the
        // surrounding loop):

        breakFrom();

    endfor;

end ExitThunks;
```

Obviously, you should avoid constructs like these in your thunks. The control flow in the procedure above is very unusual, to say the least, and others reading this code will have a difficult time fully comprehending what is going on. Of course, like other structured programming techniques that make programs easier to read, you may discover the need to write code like this under special circumstances. Just don't make a habit of doing this gratuitously.

There is a problem with breaking out of the thunks as was done in the code above: this scheme leaves a bunch of data on the stack (specifically, the thunk's parameter, the return address, and the saved EBP value in this particular example). Had *ExitThunks* pushed some registers on the stack that it needed to preserve, ESP would not be properly pointing at those register upon reaching the end of the function. Therefore, popping these registers off the stack would load garbage into the registers. Fortunately, the HLA standard exit sequence reloads ESP from EBP prior to popping EBP's value and the return address off the stack; this resynchronizes ESP prior to returning from the procedure. However, anything you push on the stack after the standard entry sequence will not be on the top of stack if you prematurely bail out of a thunk as was done in the previous example.

The only reasonable solution is to save a copy of the stack pointer's value in a local variable after you push any important data on the stack. Then restore ESP from this local (automatic) variable before attempting to pop any of that data off the stack. The following implementation of *ExitThunks* demonstrates this principle in action:

```
procedure ExitThunks;
var
    jmpFrom:        thunk;
    breakFrom:      thunk;
    ESPsave:        dword;

begin ExitThunks;

    push( eax );            // Registers we wish to preserve.
    push( ebx );
    push( ecx );
    push( edx );
    mov( esp, ESPsave );  // Preserve ESP's value for return.

    thunk jmpFrom :=
        #{
                << Code, as appropriate, for this thunk >>
```

```
            // Just jump out of this thunk and back into
            // the ExitThunks procedure:

            jmp XTlabel;
        }#;

    // Execute the thunk above (which winds up jumping to the
    // XTlabel label below:

    jmpFrom();

    XTlabel:

    // Create a loop inside the ExitThunks procedure and
    // define a thunk within this loop.  Use a BREAK statement
    // within the thunk to exit the thunk (and loop).

    forever

        thunk breakFrom :=
            #{
                << Code, as appropriate, for this thunk >>

                // Break out of this thunk and the surrounding
                // loop via the following BREAK statement:

                break;
            }#;

        // Invoke the thunk (which causes use to exit from the
        // surrounding loop):

        breakFrom();

    endfor;

    << Any other code required by the procedure >>

    // Restore ESP's value from ESPsave in case one of the thunks (or both)
    // above have prematurely exited, leaving garbage on the stack.

    mov( ESPsave, esp );

    // Restore the registers and leave:

    pop( edx );
    pop( ecx );
    pop( ebx );
    pop( eax );

end ExitThunks;
```

This scheme will work properly because the thunks always set up EBP to point at *ExitThunks*' activation record (this is true even if the program calls these thunks from some other procedures). The *ESPsave* variable must be an automatic (VAR) variable if this code is to work properly in all cases.

1.12 Handling Exceptions with Thunks

Thunks are also useful for passing exception information back to some code in the calling tree when the HLA exception handling code would be inappropriate (e.g., if you don't want to immediately abort the operation of the current code, you just want to pass data back to some previous code in the current call chain). Before discussing how to implement some exception handler with a thunk, perhaps we should discuss why we would want to do this. After all, HLA has an excellent exception handling mechanism – the TRY..ENDTRY and RAISE statements; why not use those instead of processing exceptions manually with thunks? There are two reasons for using thunks to handle exceptions – you might want to bypass the normal exception handling code (i.e., skip over TRY..ENDTRY blocks for a certain event and pass control directly to some fixed routine) or you might want to resume execution after an exception occurs. We'll look at these two mechanisms in this section.

One of the uses for thunks in exception handling code is to bypass any intermediate TRY..ENDTRY statements between the point of the exception and the handler you'd like to use for the exception. For example, suppose you have the following call chain in your program:

```
HasExceptionHandler->MayHaveOne->MayHaveAnother->CausesTheException
```

In this sequence the procedure *CausesTheException* encounters some exceptional condition. Were you to write the code using the standard RAISE and TRY..ENDTRY statements, then the last TRY..ENDTRY statement (that handles the specific exception) would execute its EXCEPT clause and deal with this exception. In the current example, that means that *MayHaveOne* or *MayHaveAnother* could trap and attempt to handle this exception. Using the standard exception handling mechanism, it is very difficult to ensure that *HasExceptionHandler* is the only procedure that responds to this exception.

One way to avoid this problem is to use a thunk to transfer control to *HasExceptionHandler* rather than the RAISE statement. By declaring a global thunk and initializing it within *HasExceptionHandler* to execute the exception handler, you can bypass any intermediate procedures in the call chain and jump directly to *HasExceptionHandler* from the offending code. Don't forget to save ESP's value and restore it if you bail out of the exception handler code inside the thunk and jump directly into the *HasExceptionHandler* code (see "Jumping Out of a Thunk" on page 1299).

Granted, needing to skip over exception handlers is a bit of a synthetic problem that you won't encounter very often in real-life programs. However, the second feature raised above, resuming the original code after handling an exception, is something you may need to do from time to time. HLA's exceptions do not allow you to resume the code that raised the exception, so if you need this capability thunks provide a good solution. To resume the interrupted code when using a thunk, all you have to do is return from the thunk in the normal fashion. If you don't want to resume the original code, then you can jump out of the thunk and into the surrounding procedure code (don't forget to save and restore ESP in that surrounding code, see "Jumping Out of a Thunk" on page 1299 for details). The nice thing about a thunk is that you don't have to decide whether you're going to bail out of the thunk or resume the execution of the original code while writing your program. You can write some code within the thunk to make this decision at run-time.

1.13 Using Thunks in an Appropriate Manner

This chapter presents all sorts of novel uses for thunks. Thunks are really neat and you'll find all kinds of great uses for them if you just think about them for a bit. However, it's also easy to get carried away and use thunks in an inappropriate fashion. Remember, thunks are not only a pointer to a procedure but a pointer to an execution environment as well. In many circumstances you don't need the execution environment pointer (i.e., the pointer to the activation record). In those cases you should remember that you can use a simple procedure pointer rather than a thunk to indirectly call the "thunk" code. A simple indirect call is a bit more efficient than a thunk invocation, so unless you really need all the features of the thunk, just use a procedure pointer instead.

1.14 Putting It All Together

Although thunks are quite useful, you don't see them used in many programs. There are two reasons for this – most high level languages don't support thunks and, therefore, few programmers have sufficient experience using thunks to know how to use this appropriately. Most people learning assembly language, for example, come from a standard imperative programming language background (C/C++, Pascal, BASIC, FORTRAN, etc.) and have never seen this type of programming construct before. Those who are used to programming in languages where thunks are available (or a similar construct is available) tend not to be the ones who learn assembly language.

If you happen to lack the prerequisite knowledge of thunks, you should not write off this chapter as unimportant. Thunks are definitely a programming tool you should be aware of, like recursion, that's really handy in lots of situations. You should watch out for situations where thunks are applicable and use them as appropriate.

We'll see additional uses for thunks in the next chapter on iterators and in the chapter on advanced parameter passing techniques, later in this volume.

Iterators Chapter Two

2.1 Chapter Overview

This chapter discusses the low-level implementation of iterators. Earlier, this text briefly discussed iterators and the FOREACH loop in the chapter on intermediate procedures (see “Iterators and the FOREACH Loop” on page 843). This chapter will review that information, discuss some uses for iterators, and then present the low-level implementation of this interesting control structure.

2.2 Review of Iterators

An iterator is a cross between a control structure and a function. Although common high level languages do not support iterators, they are present in some very high level languages[1]. Iterators provide a combination state machine/function call mechanism that lets a function pick up where it last left off on each new call. Iterators are also part of a loop control structure, with the iterator providing the value of the loop control variable on each iteration.

To understand what an iterator is, consider the following for loop from Pascal:

```
for I := 1 to 10 do <some statement>;
```

When learning Pascal you were probably taught that this statement initializes *i* with one, compares *i* with 10, and executes the statement if *i* is less than or equal to 10. After executing the statement, the *FOR* statement increments *i* and compares it with 10 again, repeating the process over and over again until *i* is greater than 10.

While this description is semantically correct, and indeed, it’s the way that most Pascal compilers implement the FOR loop, this is not the only point of view that describes how the for loop operates. Suppose, instead, that you were to treat the TO reserved word as an operator. An operator that expects two parameters (one and ten in this case) and returns the range of values on each successive execution. That is, on the first call the TO operator would return one, on the second call it would return two, etc. After the tenth call, the TO operator would fail which would terminate the loop. This is exactly the description of an iterator.

In general, an iterator controls a loop. Different languages use different names for iterator controlled loops, this text will just use the name FOREACH as follows:

```
foreach iterator() do
    statements;
endfor;
```

An iterator returns two values: a boolean success or failure value and a function result. As long as the iterator returns success, the FOREACH statement executes the statements comprising the loop body. If the iterator returns failure, the FOREACH loop terminates and executes the next sequential statement following the FOREACH loop’s body.

Iterators are considerably more complex than normal functions. A typical function call involves two basic operations: a call and a return. Iterator invocations involve four basic operations:

1) Initial iterator call
2) Yielding a value
3) Resumption of an iterator

1. Ada and PL/I support very limited forms of iterators, though they do not support the type of iterators found in CLU, SETL, Icon, and other languages.

4) Termination of an iterator.

To understand how an iterator operates, consider the following short example:

```
iterator range( start:int32; stop:int32 );
begin range;

    forever

        mov( start, eax );
        breakif( eax > stop );
        yield();
        inc( start );

    endfor;

end range;
```

In HLA, iterator calls may only appear in the FOREACH statement. With the exception of the "yield();" statement above, anyone familiar with HLA should be able to figure out the basic logic of this iterator.

An iterator in HLA may return to its caller using one of two separate mechanisms, it can return to the caller by exiting through the "end Range;" statement or it may yield a value by executing the "yield();" statement. An iterator succeeds if it executes the "yield();" statement, it fails if it simply returns to the caller. Therefore, the FOREACH statement will only execute its corresponding statement if you exit an iterator with a "yield();". The FOREACH statement terminates if you simply return from the iterator. In the example above, the iterator returns the values *start..stop* via a "yield();" statement and then the iterator terminates. The loop

```
foreach Range(1,10) do

    stdout.put( (type uns32 eax ), nl );

endfor;
```

is comparable to the code:

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

    stdout.put( (type uns32 eax ), nl );

endfor;
```

When an HLA program first executes the FOREACH statement, it makes an initial call to the iterator. The iterator runs until it executes a "yield();" or it returns. If it executes the "yield();" statement, it returns the value in EAX as the iterator result and it succeeds. If it simply returns, the iterator returns failure and no iterator result. In the current example, the initial call to the iterator returns success and the value one.

Assuming a successful return (as in the current example), the FOREACH statement returns the current result in EAX and executes the FOREACH loop body. After executing the loop body, the FOREACH statement calls the iterator again. However, this time the FOREACH statement resumes the iterator rather than making an initial call. An iterator resumption continues with the first statement following the last "yield();" it executes. In the *range* example, a resumption would continue execution at the "inc(start);" statement. On the first resumption, the *range* iterator would add one to *start*, producing the value two. Two is less than ten (*stop's* value) so the FOREACH loop would repeat and the iterator would yield the value two. This process would repeat over and over again until the iterator yields ten. Upon resuming after yielding ten, the iterator would increment start to eleven and then return, rather than yield, since this new value is not less than or equal to ten. When the *Range* iterator returns (fails), the FOREACH loop terminates.

2.2.1 Implementing Iterators Using In-Line Expansion

The implementation of an iterator is rather complex. To begin with, consider a first attempt at an assembly implementation of the FOREACH statement above:

```
       push( 1 );    // Manually pass 1 and 10 as parameters.
       push( 10 );
       call Range_initial;
       jc Failure;
ForLp: stdout.put( (type uns32 eax), nl );
       call Range_Resume;
       jnc ForLp;
Failure:
```

Although this looks like a straight-forward implementation project, there are several issues to consider. First, the call to *Range_Resume* above looks simple enough, but there is no fixed address that corresponds to the resume address. While it is certainly true that this *Range* example has only one resume address, in general you can have as many "yield();" statements as you like in an iterator. For example, the following iterator returns the values 1, 2, 3, and 4:

```
iterator OneToFour;
begin OneToFour;

       mov( 1, eax ); yield();
       mov( 2, eax ); yield();
       mov( 3, eax ); yield();
       mov( 4, eax ); yield();

end OneToFour;
```

The initial call would execute the "mov(1, eax);" and "yield();" statements. The first resumption would execute the "mov(2, eax);" and "yield();" statements, the second resumption would execute "mov(3, eax);" and "yield();", etc. Obviously there is no single resume address the calling code can count on.

There are a couple of additional details left to consider. First, an iterator is free to call procedures and functions. If such a procedure or function executes the "yield();" statement then resumption by the FOREACH statement continues execution within the procedure or function that executed the "yield();"[2]. Second, the semantics of an iterator require all local variables and parameters to maintain their values until the iterator terminates. That is, yielding does not deallocate local variables and parameters. Likewise, any return addresses left on the stack (e.g., the call to a procedure or function that executes the "yield();" statement) must not be lost when a piece of code yields and the corresponding FOREACH statement resumes the iterator. In general, this means you cannot use the standard call and return sequence to yield from or resume to an iterator because you have to preserve the contents of the stack.

While there are several ways to implement iterators in assembly language, perhaps the most practical method is to have the iterator call the loop controlled by the iterator and have the loop return back to the iterator function. Of course, this is counter-intuitive. Normally, one thinks of the iterator as the function that the loop calls on each iteration, not the other way around. However, given the structure of the stack during the execution of an iterator, the counter-intuitive approach turns out to be easier to implement.

Some high level languages support iterators in exactly this fashion. For example, Metaware's Professional Pascal Compiler for the PC supports iterators[3]. Were you to create a Professional Pascal code sequence as follows:

```
iterator OneToFour:integer;
begin
   yield 1;
```

2. This requires the use of nested procedures, a subject we will discuss in a later chapter.
3. Obviously, this is a non-standard extension to the Pascal programming language provided in Professional Pascal.

```
        yield 2;
        yield 3;
        yield 4;
    end;
```

and call it in the main program as follows:

```
    for i in OneToFour do writeln(i);
```

Professional Pascal would completely rearrange your code. Instead of turning the iterator into an assembly language function and calling this function from within the FOR loop body, this code would turn the FOR loop body into a function, expand the iterator in-line (much like a macro) and call the FOR loop body function on each yield. That is, Professional Pascal would probably produce assembly language that looks something like the following:

```
// The following procedure corresponds to the for loop body
// with a single parameter (I) corresponding to the loop
// control variable:

procedure ForLoopCode( i:int32 ); nodisplay;
begin ForLoopCode;

    mov( i, eax );
    stdout.put( i, nl );

end ForLoopCode;

// The follow code would be emitted in-line upon encountering the
// for loop in the main program, it corresponds to an in-line
// expansion of the iterator as though it were a macro,
// substituting a call for the yield instructions:

        ForLoopCode( 1 );
        ForLoopCode( 2 );
        ForLoopCode( 3 );
        ForLoopCode( 4 );
```

This method for implementing iterators is convenient and produces relatively efficient (fast) code. It does, however, suffer from a couple drawbacks. First, since you must expand the iterator in-line wherever you call it, much like a macro, your program could grow large if the iterator is not short and you use it often. Second, this method of implementing the iterator completely hides the underlying logic of the code and makes your assembly language programs difficult to read and understand.

2.2.2 Implementing Iterators with Resume Frames

In-line expansion is not the only way to implement iterators. There is another method that preserves the structure of your program at the expense of a slightly more complex implementation. Several high level languages, including Icon and CLU, use this implementation.

To start with, you will need another stack frame: the *resume frame*. A resume frame contains two entries: a yield return address (that is, the address of the next instruction after the yield statement) and a *dynamic link*, that is a pointer to the iterator's activation record. Typically the dynamic link is just the value in the EBP register at the time you execute the yield statement. This version implements the four parts of an iterator as follows:

1) A CALL instruction for the initial iterator call,

2) A CALL instruction for the YIELD statement,

3) A RET instruction for the resume operation, and

4) A RET instruction to terminate the iterator.

To begin with, an iterator will require two return addresses rather than the single return address you would normally expect. The first return address appearing on the stack is the termination return address. The second return address is where the subroutine transfers control on a *yield* operation. The calling code must push these two return addresses upon initial invocation of the iterator. The stack, upon initial entry into the iterator, should look something like Figure 2.1.

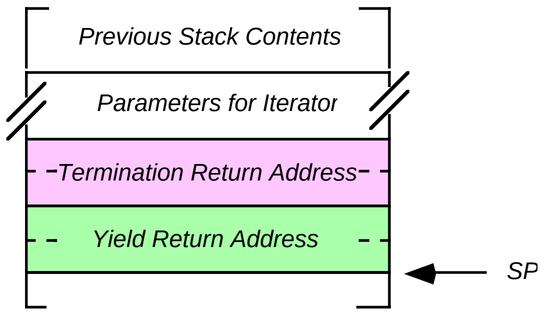

Figure 2.1 Iterator Activation Record

As an example, consider the *Range* iterator presented earlier. This iterator requires two parameters, a starting value and an ending value:

```
foreach range(1,10) do

    stdout.put( i, nl );

endfor;
```

The code to make the initial call to the *range* iterator, producing a stack like the one above, could be the following:

```
        push( 1 );         // Push start parameter value.
        push( 10 );        // Push stop parameter value.
        push( &ForDone);   // Push termination address.
        call range;        // Call the iterator.
            .
            .
            .
ForDone:
```

fordone is the first statement immediately following the FOREACH loop, that is, the instruction to execute when the iterator returns failure. The FOREACH loop body must begin with the first instruction following the call to *range*. At the end of the FOREACH loop, rather than jumping back to the start of the loop, or calling the iterator again, this code should just execute a RET instruction. The reason will become clear in a moment. So the implementation of the above FOREACH statement could be the following:

```
        push( 1 );              // Push start parameter value.
        push( 10 );             // Push stop parameter value.
        push( &ForDone);        // Push termination address.
        call range;             // Call the iterator.
        push( ebp );            // Preserve iterator’s ebp value.
        mov( [esp+8], ebp );    // Get original EBP value passed to us by range.
        stdout.put( i, nl );    // Display i’s value.
        pop( ebp );             // Restore iterator’s EBP value.
        ret(4 );                // Return and clean EBP value off stack.
ForDone:
```

Granted, this doesn’t look anything at all like a loop. However, by playing some major tricks with the stack, you’ll see that this code really does iterate the loop body (*stdout.put*) as intended.

Now consider the *range* iterator itself, here’s the (mostly) low-level code to do the job:

```
iterator range( start:int32; stop:int32 ); @nodisplay; @noframe;
begin range;

    push( ebp );      // Standard Entry Sequence
    mov( esp, ebp );

    ForEverLbl:

        mov( start, eax );
        cmp( eax, stop );
        jng ForDone;

        yield();
        inc( start );
        jmp ForEverLbl;

    ForDone:
        pop( ebp );
        add( 4, esp );
        ret( 8 );

end range;
```

Although this routine is rather short, don’t let its size deceive you; it’s quite complex. The best way to describe how this iterator operates is to take it a few instructions at a time. The first two instructions are the standard entry sequence for a procedure. Upon execution of these two instructions, the stack looks like that in Figure 2.2.

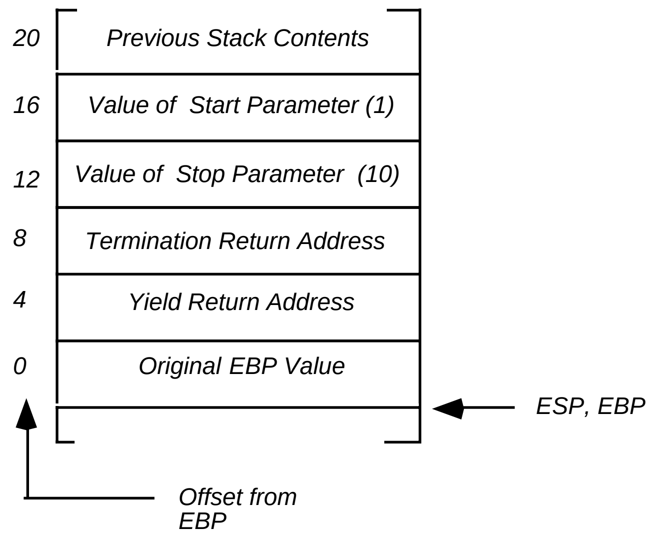

Figure 2.2 Range Activation Record

The next three statements in the *range* iterator, at label *ForEverLbl*, implement the termination test of the loop. When the *start* parameter contains a value greater than the *stop* parameter, control transfers to the *ForDone* label at which point the code pops the value of EBP off the stack, pops the success return address off the stack (since this code will not return back to the body of the iterator loop) and then returns via the termination return address that is immediately above the success return address on the stack. The return instruction also pops the two parameters (eight bytes) off the stack.

The real work of the iterator occurs in the body of the loop. The main question here is "what is this *yield* procedure and what is it doing?". To understand what *yield* is, we must consider what it is that *yield* does. Whenever the iterator executes the *yield* statement, it calls the body of the FOREACH loop that invoked the iterator. Since the body of the FOREACH loop is the first statement following the call to the iterator, it turns out that the iterator's return address points at that body of code. Therefore, the *yield* statement does the unusual operation of calling a subroutine pointed at by the iterator's (success) return address.

Simply calling the body of the FOREACH loop is not all the *yield* call must do. The body of the FOREACH loop is (probably) in a different procedure with its own activation record. That FOREACH loop body may very well access variables that are local to the procedure containing that loop body; therefore, the *yield* statement must also pass the original procedure's EBP value as a parameter so that the loop body can restore EBP to point at the FOREACH loop body's activation record (while, of course, preserving EBP's value within the iterator itself). The caller's EBP value (also known as the dynamic link) was the value the iterator pushes on the stack in the standard entry sequence. Therefore, [EBP+0] will point at this dynamic link value. To properly implement the yield operation, the iterator must emit the following code:

```
push( ebp );                // Save iterator's activation record pointer.
call((type dword [ebp+4])); // Call the return address.
```

The PUSH and CALL instructions build the resume frame and then return control to the body of the FOREACH loop. The CALL instruction is not calling a subroutine. What it is really doing here is finishing off the resume frame (by storing the *yield* resume address into the resume frame) and then it returns control back to the body of the FOREACH loop by jumping indirect through the success return address pushed on the stack by the initial call to the iterator. After the execution of this call, the stack frame looks like that in Figure 2.3.

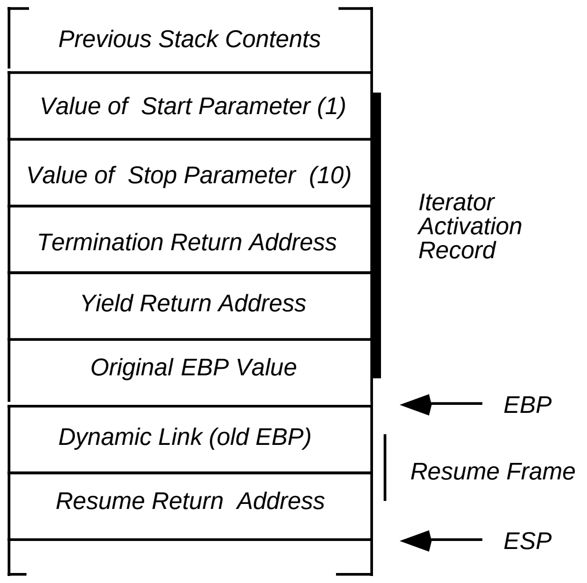

Figure 2.3 Range Resume Record

By convention, the EAX register contains the return value for the iterator. As with functions, EAX is a good place to return the iterator return result.

Immediately after yielding back to the FOREACH loop, the code must reload EBP with the original value prior to the iterator invocation. This allows the calling code to correctly access parameters and local variables in its own activation record rather than the activation record of the iterator. The FOREACH loop body begins by preserving EBP's value (the pointer to iterator's activation record) and then loading EBP with the value pushed on the stack by the *yield* statement. Of course, in this example reloading EBP isn't necessary because the body of the FOREACH loop does not reference any memory locations off the EBP register, but in general you will need to save EBP's value and load EBP with the value pushed on the stack by the yield statement.

At the end of the FOREACH loop body the "pop(ebp);" and "ret(4);" instructions restore EBP's value, cleans up the environment pointer passed as a parameter, and resumes the iterator. The RET instruction pops the return address off the stack which returns control back to the iterator immediately after the call emitted by the *yield* statement.

Of course, this is a lot of work to create a piece of code that simply repeats a loop ten times. A simple FOR loop would have been much easier and quite a bit more efficient that the FOREACH implementation described in this section. This section used the *range* iterator because it was easy to show how iterators work using *range*, not because actually implementing *range* as an iterator is a good idea.

Note that HLA does not provide an actual *yield* statement. If you look carefully at this code, and you think back to the last chapter, you will notice that the *yield* "statement" generates exactly the same code as a thunk invocation. Indeed, were you to dump the HLA symbol table when compiling a program containing the *range* iterator, you'd discover that *yield* is actually a local variable of type thunk within the *range* iterator code. The offset of this thunk is zero (from EBP) in the iterator's activation record (see Figure 2.4):

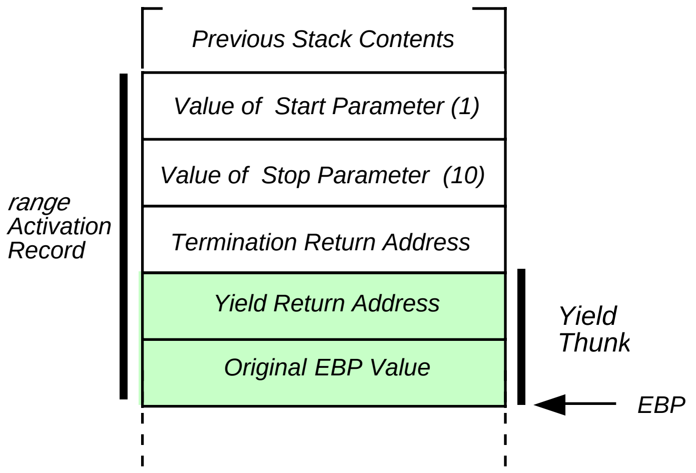

Figure 2.4 Yield Thunk in the range Iterator's Activation Record

This thunk value is somewhat unusual. If you look closely, you'll realize that this thunk's value is actually the return address the original call to *range* pushes along with the old EBP value that the *range* iterator code pushes as part of the standard entry sequence. In other words, the call to *range* and the standard entry sequence automatically initializes the *yield* thunk! How's that for elegance? All the HLA compiler has to do to create the *yield* thunk is to automatically create this "yield" symbol and associate *yield* with the address of the old EBP value that the *range* standard entry sequence pushes on the stack.

As you may recall from the chapter on Thunks, the calling sequence for a thunk is the following:

```
push( << Thunk's Environment Pointer>> );
call( << Thunk's Code Pointer >> );
```

In the case of the *yield* thunk in an iterator, the calling sequence looks like this:

```
pushd( [ebp] );               // Pass iterator caller's EBP as parameter.
call( (type dword [ebp+4] )); // Call the FOREACH loop's body.
```

The body of the FOREACH loop, like any thunk, must preserve EBP's value and load EBP with the value pushed on the stack in the code sequence above. Prior to returning (resuming) back to the iterator, the FOREACH loop body must restore the iterator's EBP value and it must remove the environment pointer parameter from the stack upon return. The code given earlier for the FOREACH loop does this:

```
// "FOREACH range( 1, 10) do"  statement:

    push( 1 );              // Push start parameter value.
    push( 10 );             // Push stop parameter value.
    push( &ForDone);        // Push termination address.
    call range;             // Call the iterator.

// FOREACH loop body (a thunk):

    push( ebp );            // Preserve iterator's ebp value.
    mov( [esp+8], ebp );    // Get original EBP value passed to us by range.
    stdout.put( i, nl );    // Display i's value.
    pop( ebp );             // Restore iterator's EBP value.
    ret(4 );                // Return and clean EBP value off stack.
    // endfor;

ForDone:
```

Of course, HLA does not make you write this low-level code. You can actually use a FOREACH statement in your program. The above code is the low-level implementation of the following high-level HLA code:

```
    foreach range( 1, 10 ) do

        stdout.put( i, nl );

    endfor;
```

The HLA compiler automatically emits the code to preserve and set up EBP at the beginning of the FOREACH loop's body; HLA also automatically emits the code to restore EBP and return to the iterator (removing the environment pointer parameter from the stack).

2.3 Other Possible Iterator Implementations

Thus far, this text has given two different implementations for iterators and the FOREACH loop. Although the resume frame/thunk implementation of the previous section is probably the most common implementation in HLA programs (since the HLA compiler automatically generates this type of code for FOREACH loops and iterators), don't get the impression that this is the only, or best, way to implement iterators and the FOREACH loop. Other possible implementations certainly exist and in some specialized situations some other implementation may offer some advantages. In this section we'll look at a couple of ways to implement iterators and the FOREACH loop.

The standard HLA implementation of iterators uses two separate return addresses for an iterator call: a success/yield address and a failure address. This organization is elegant given the thunk implementation of HLA's FOREACH statement, but there are other ways to return success/failure from an iterator. For example, you could use the value of the carry flag upon return from the iterator call to denote success or failure. Then a call to the iterator might take the following form:

```
ForEachLoopLbl:
        << Push any Necessary Parameters >>
        call iter;
        jc iterFails;

        << Code for the body of the iterator >>

        jmp ForEachLoopLbl;

iterFails:
```

One problem with this approach is that the code reenters the iterator on each iteration of the loop. This means it always passes the same parameters and reconstructs the activation record on each call of the iterator. Clearly, you cannot use this scheme if the iterator needs to maintain state information in its activation record between calls to the iterator. Furthermore, if the iterator yields from different points, then transferring control to the first statement after each yield statement will be a problem. There is a trick you can pull to tell the iterator whether this is the first invocation or some other invocation - pass a special parameter to indicate the first call of an iterator. You can do this as follows:

```
        pushd( 0 );   // Zero indicates the first call to iter.
ForEachLoopLbl:
        << Push Any Other Necessary Parameters >>
        call iter;      // iter is a standard HLA procedure, not an iterator.
        jc iterFails;
        << Code for the body of the iterator >>

        pushd( 1 ); // One indicates a re-entry into the iterator.
        jmp ForEachLoopLbl;

iterFails:
```

Notice how this code pushes a zero as the first parameter on the first call to *iter* and it pushes a one on each invocation thereafter.

What if the iterator needs to maintain state information (i.e., local variable values) between calls? Well, the easiest way to handle this using the current scheme is to pass extra parameters by value and use those parameters as the local variables. When the iterator returns success, it should not clean up the parameters on the stack, instead, it will leave them there for the next iteration of the "FOREACH" loop. E.g., consider the following implementation and invocation of the *ArithRange* iterator (this iterator returns the sum of all the values between the *start* and *stop* parameters):

```
// Note: we have to use a procedure, not an iterator, here because we don’t
// want HLA generating funny code for us.
//
// "sum" is actually a local variable in which we maintain state information.
// It must contain zero on the first entry to denote the intial entry into
// this code.

procedure ArithRange( start:uns32; stop:uns32; sum:uns32 );
    @nodisplay;  @noframe;
begin ArithRange;

    push( ebp );          // Standard entry sequence.
    mov( esp, ebp );


    mov( start, eax );
    if( eax <= stop ) then

        add( sum, eax );  // Compute arithmetic sum of values.
        mov( eax, sum );  // Save for the next time through and return in EAX.
        pop( ebp );       // Restore pointer to caller’s environment.
        clc;              // Indicate success on return.
        ret();            // Note that we don’t pop parameters!

    endif;
    pop( ebp );          // Restore pointer to caller’s environment.
    stc;                 // Indicate return on failure.
    ret( 12 );           // On failure, remove the parameters from the stack.

end ArithRange;
        .
        .
        .
        pushd( 1 );        // Pass the start parameter value.
        pushd( 10 );       // Pass the stop parameter value.
        pushd( 0 );        // Must pass zero in for sum value.
ForEachLoop:
        call ArithRange;   // Call our "iterator".
        jc ForEachDone;    // Quit on failure, fall through on success.

        << Foreach loop body code >>

        jmp ForEachLoop;
```

```
ForEachDone:
```

Notice how this code pushes the parameters on the stack for the first invocation of the ArithRange iterator, but it does not push the parameters on the stack on successive iterations of the loop. That's because the code does not remove these parameter values until it fails. Therefore, on each iteration of the loop, the parameter values left on the stack by the previous invocation of *ArithRange* are still on the stack for the next invocation. When the iterator fails, it pops the parameters off the stack so the stack is clean on exit from the "FOREACH" loop above.

If you can spare a register, there is a slightly more efficient way to implement iterators and the FOREACH loop (HLA doesn't use this scheme by default because it promise not to monkey with your register set in the code generation for the high level control structures). Consider the following code that HLA emits for a *yield* call:

```
push( [ebp] );        // Pass FOREACH loop’s EBP value as a parameter.
call( [ebp+4] );      // Call the success address.
```

The FOREACH loop body code looks like the following:

```
push( ebp );          // Save iterator’s EBP value.
mov( [esp+8], ebp );  // Fetch our EBP value pushed by the iterator.
<< loop body >>
pop( ebp );           // Restore iterator’s EBP value.
ret( 4 );             // Resume the iterator and remove EBP value.
```

This code can be improved slightly by preserving and setting the EBP value within the iterator. Consider the following yield and FOREACH loop body code:

```
push( ebp );          // Save iterator’s EBP value.
mov( [ebp+4], edx );  // Put success address into an available register.
mov( [ebp], ebp );    // Set up FOREACH loop’s EBP value.
call edx;             // Call the success address.
pop( ebp );           // Restore our EBP value.
```

Here's the corresponding FOREACH loop body:

```
<< Loop Body >>
ret();
```

These scheme isn't amazingly better than the standard resume frame approach (indeed, it is about the same). But in some situations the fact that the loop body code doesn't have to mess with the stack may be important.

2.4 Breaking Out of a FOREACH Loop

The BREAK, BREAKIF, CONTINUE, and CONTINUEIF statements are active within a FOREACH loop, but there are some problems you must consider if you attempt to break out of a FOREACH loop with a BREAK or BREAKIF statement. In this section we'll look at the problems of prematurely leaving a FOREACH loop.

Keep in mind that an iterator leaves some information on the stack during the execution of the FOREACH loop body. Remember, the iterator doesn't return back to the FOREACH loop in order to execute the loop body; it actually calls the FOREACH loop body. That call leaves a resume frame plus all parameters, local variables, and other information in the iterator's activation record on the stack when it calls the FOREACH loop body. If you attempt to bail out of the FOREACH loop using BREAK, BREAKIF, or (worse still) a conditional or unconditional jump, ESP does not automatically revert back to the value prior to the execution of the FOREACH statement. The HLA generated code can only clean up the stack properly if the iterator returns via the failure address.

Although HLA cannot clean up the stack for you, it is quite possible for you to clean up the stack yourself. The easiest way to do this is to store the value in ESP to a local variable immediately prior to the execution of the FOREACH statement. Then simply reload ESP from this value prior to prematurely leaving the FOREACH loop. Here's some code that demonstrates how to do this:

```
mov( esp, espSave );
foreach range( 1, 10 ) do

    << foreach loop body >>

    if( some_condition ) then

        mov( espSave, esp );
        break;

    endif;

    << more loop body code >>

endfor;
```

By restoring ESP from the *espSave* variable, this code removes all the activation record information from the stack prior to leaving the FOREACH loop body. Notice the MOV instruction immediately before the FOREACH statement that saves the stack position prior to calling the *range* iterator.

2.5 An Iterator Implementation of the Fibonacci Number Generator

Consider for a moment the Fibonacci number generator from the chapter on Thunks (this is the slow, non-thunk, implementation):

```
// Standard fibonacci function using the slow recursive implementation.

procedure slowfib( n:uns32 ); nodisplay; returns( "eax" );
begin slowfib;

    // For n= 1,2 just return 1.

    if( n <= 2 ) then

        mov( 1, eax );

    else

        // Return slowfib(n-1) + slowfib(n-2) as the function result:

        dec( n );
        slowfib( n );   // compute fib(n-1)
        push( eax );    // Save fib(n-1);

        dec( n );       // compute fib(n-2);
        slowfib( n );

        add( [esp], eax );  // Compute fib(n-1) [on stack] + fib(n-2) [in eax].
        add( 4, esp );      // Remove old value from stack.

    endif;
```

```
end slowfib;
```

Program 2.1 Recursive Implementation of the Fibonacci Number Generator

This particular function generates the nth Fibonacci number by computing the first through the nth the Fibonacci number. If you wanted to generate a sequence of Fibonacci numbers, you could use a FOR loop as follows:

```
for( mov( 1, ebx ); ebx < n; inc( ebx )) do

    slowfib( ebx );
    stdout.put( "Fib(", (type uns32 ebx), ") = ", (type uns32 eax), nl );

endfor;
```

True to its name, this implementation of *slowfib* runs quite slowly as *n* gets larger. The reason this function takes so much time (as pointed out in the chapter on Thunks) is that each recursive call recomputes all the previous Fibonacci numbers. Given the (n-1)st and (n-2)nd Fibonacci numbers, computing the nth Fibonacci number is trivial and very efficient – you simply add the previous two values together. The efficiency loss occurs when the recursive implementation computes the same value over and over again. The chapter on Thunks described how to eliminate this recomputation by passing the computed values to other invocations of the functions. This allows the Fibonacci function to compute the nth Fibonacci number in *n* units of time rather than 2^n units of time, a dramatic improvement. However, since each call to the (efficient) Fibonacci generator requires *n* units of time to compute its result, the loop above (which repeats n times) requires approximately n^2 units of time to run. This does not seem reasonable since it clearly takes only a few instructions to compute a new Fibonacci number given the previous two values; that is, we should be able to compute the nth Fibonacci number in *n* units of time. Iterators provide a trivial way to implement a Fibonacci number generator that generates a sequence of *n* Fibonacci numbers in *n* units of time. The following program demonstrates how to do this.

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

// Fibonocci function using a thunk to calculate fib(n-2)
// without making a recursive call.

procedure fib( n:uns32; nm2:thunk ); nodisplay; returns( "eax" );
var
    n2: uns32;      // A recursive call to fib stores fib(n-2) here.
    t:  thunk;      // This thunk actually stores fib(n-2) in n2.

begin fib;

    // Special case for n = 1, 2.  Just return 1 as the
    // function result and store 1 into the fib(n-2) result.

    if( n <= 2 ) then

        mov( 1, eax );  // Return as n-1 value.
        nm2();          // Store into caller as n-2 value.

    else

        // Create a thunk that will store the fib(n-2) value
        // into our local n2 variable.
        thunk   t :=
                #{
                    mov( eax, n2 );
                }#;

        mov( n, eax );
        dec( eax );
        fib( eax, t );  // Compute fib(n-1).

        // Pass back fib(n-1) as the fib(n-2) value to a previous caller.

        nm2();


        // Compute fib(n) = fib(n-1) [in eax] + fib(n-2) [in n2]:

        add( n2, eax );

    endif;

end fib;


// Standard fibonocci function using the slow recursive implementation.

procedure slowfib( n:uns32 ); nodisplay; returns( "eax" );
begin slowfib;

    // For n= 1,2 just return 1.

    if( n <= 2 ) then

        mov( 1, eax );

    else

        // Return slowfib(n-1) + slowfib(n-2) as the function result:

        dec( n );
        slowfib( n );   // compute fib(n-1)
        push( eax );    // Save fib(n-1);

        dec( n );       // compute fib(n-2);
        slowfib( n );

        add( [esp], eax );  // Compute fib(n-1) [on stack] + fib(n-2) [in eax].
        add( 4, esp );      // Remove old value from stack.

    endif;

end slowfib;


// FibNum-
//
//  Iterator that generates all the fibonacci numbers between 1 and n.

iterator FibNum( n:uns32 ); nodisplay;
var
    Fibn_1: uns32;      // Holds Fib(n-1) for a given n.
```

```
        Fibn_2: uns32;      // Holds Fib(n-2) for a given n.
        CurFib: uns32;      // Current index into fib sequence.

    begin FibNum;

        mov( 1, Fibn_1 );   // Initialize these guys upon initial entry.
        mov( 1, Fibn_2 );
        mov( 1, eax );      // Fib(0) = 1
        yield();
        mov( 1, eax );      // Fib(1) = 1;
        yield();
        mov( 2, CurFib );
        forever

            mov( CurFib, eax );     // Compute sequence up to the nth #.
            breakif( eax > n );
            mov( Fibn_2, eax );     // Compute this result.
            add( Fibn_1, eax );

            // Recompute the Fibn_1 and Fibn_2 values:

            mov( Fibn_1, Fibn_2 );
            mov( eax, Fibn_1 );

            // Return current value:

            yield();

            // Next value in sequence:

            inc( CurFib );

        endfor;

    end FibNum;



    var
        prevTime:dword[2];      // Used to hold 64-bit result from RDTSC instr.
        qw: qword;              // Used to compute difference in timing.
        dummy:thunk;            // Used in original calls to fib.

    begin fibIter;

        // "Do nothing" thunk used by the initial call to fib.
        // This thunk simply returns to its caller without doing
        // anything.

        thunk dummy := #{ }#;


        // Call the fibonocci routines to "prime" the cache:

        fib( 1, dummy );
        slowfib( 1 );
        foreach FibNum( 1 ) do
        endfor;
```

```
// Okay, compute the running times for the three fibonocci routines to
// generate a sequence of n fibonacci numbers where n ranges from
// 1 to 32:

for( mov( 1, ebx ); ebx < 32; inc( ebx )) do

    // Emit the index:

    stdout.put( (type uns32 ebx):2, stdio.tab );

    // Compute the # of cycles needed to compute the Fib via iterator:

    rdtsc();
    mov( eax, prevTime );
    mov( edx, prevTime[4] );

    foreach FibNum( ebx ) do

    endfor;

    rdtsc();
    sub( prevTime, eax );
    sbb( prevTime[4], edx );
    mov( eax, (type dword qw));
    mov( edx, (type dword qw[4]));

    stdout.putu64Size( qw, 4, ' ' );
    stdout.putc( stdio.tab );




    // Read the time stamp counter before calling fib:

    rdtsc();
    mov( eax, prevTime );
    mov( edx, prevTime[4] );

    for( mov( 1, ecx ); ecx <= ebx; inc( ecx )) do

        fib( ecx, dummy );

    endfor;

    // Read the timestamp counter and compute the approximate running
    // time of the current call to fib:

    rdtsc();
    sub( prevTime, eax );
    sbb( prevTime[4], edx );
    mov( eax, (type dword qw));
    mov( edx, (type dword qw[4]));

    // Display the results and timing from the call to fib:

    stdout.putu64Size( qw, 10, ' ' );
    stdout.putc( stdio.tab );


    // Okay, repeat the above for the slowfib implementation:
            rdtsc();
            mov( eax, prevTime );
            mov( edx, prevTime[4] );

            for( mov( 1, ecx ); ecx <= ebx; inc( ecx )) do

                slowfib( ebx );

            endfor;

            rdtsc();
            sub( prevTime, eax );
            sbb( prevTime[4], edx );
            mov( eax, (type dword qw));
            mov( edx, (type dword qw[4]));

            stdout.putu64Size( qw, 10, ' ' );
            stdout.newln();



        endfor;

    end fibIter;
```

Program 2.2 Fibonacci Iterator Example Program

The important concept here is that the *FibNum* iterator maintains its state across calls. In particular, it keeps track of the current iteration and the previous two Fibonacci values. Therefore, the iterator takes very little time to compute the result of each number in the sequence. This is far more efficient than either Fibonacci number generator from the chapter on Thunks, as the following table attests.

Table 1: CPU Cycle Times for Various Fibonacci Implementations

n	Iterator Implementation	Thunk Implementation	Recursive Implementation
1	156	233	98
2	148	98	77
3	178	221	271
4	193	376	399
5	213	509	879
6	213	712	1758
7	233	919	3493
8	252	1166	6531
9	271	1460	12568

Table 1: CPU Cycle Times for Various Fibonacci Implementations

n	Iterator Implementation	Thunk Implementation	Recursive Implementation
10	290	1759	22871
11	308	2139	40727
12	331	2503	71986
13	349	2938	126443
14	372	3341	220673
15	380	3877	382364
16	402	4326	660506
17	417	4977	1160660
18	452	5528	1954253
19	487	6222	3322819
20	479	6840	5685066
21	524	7691	9621772
22	545	8313	16339720
23	576	9292	27709571
24	599	10029	47036825
25	616	11274	80102556
26	650	12348	132583731
27	653	13172	222580780
28	683	14339	374788752
29	694	15394	627559062
30	722	16363	1054201515
31	732	17727	1756744511

2.6 Iterators and Recursion

It is completely possible, and sometimes very useful, to recursively call an iterator. In this section we'll explore the syntax for this and present a couple of useful recursive iterators.

Although there is nothing stopping you from manually calling an iterator with the CALL instruction, the only valid (high level syntax) invocation of an iterator is via the FOREACH statement. Therefore, to recursively call an iterator, that iterator must contain a FOREACH loop to recursively call itself.

Iterators are especially useful for traversing tree and graph data structures. Some of the best (and most efficient) examples of recursive iterators are those that traverse such structures. Unfortunately, this text does not assume the prerequisite knowledge of such data structures, so it cannot use such examples to demonstrate recursive iterators. Nevertheless, it's worth mentioning this fact here because if you are familiar with graph traversal algorithms (or will be learning them in the future) you should consider using iterators for this purpose.

One useful iterator that doesn't require a tremendous amount of prerequisite knowledge is the traversal of a binary search tree implemented within an array. We won't go into the details of what a binary search tree is or why you would use it here other than to describe some properties of that tree. A binary search tree, implemented as an array, is a data structure that allows one to quickly search for some value within the structure. The values are arranged in the tree such that after each comparison you can eliminate half of the possible values with a single comparison. As a result, if the array contains *n* items, you can locate a particular item of interest in $\log_2 n$ units of time. To achieve this efficient search time, you have to arrange the data in the array in a particular fashion and then use a specific algorithm when searching through the tree. For the sake of our example, we'll assume that the data in the array is sorted (that is, a[0] < a[1] < a[2] < ... < a[n-1] for some definition of "less than"). The binary search algorithm then takes the following form:

```
1.    Set i = n
2.    Set j = i / 2  (integer/truncating division)
3.    Quit if i=j  (failed to find value in the search tree).
4.    Compare the key value (the one you're searching for) against a[i].
5.    If key > a[i] then set j = (i - j + 1)/2 + j
      else set i = j and then  j = i/2
6.    Go to step 3.
```

How this works and what it does is irrelevant here. What is important to this section is the arrangement of the data in the array that forms the binary search tree (specifically, the sorted nature of the data). Of course, if we wanted to generate a list of numbers in the sorted order, that would be especially trivial, all we would have to do is step through the array one element at a time. Suppose, however, that we wanted to generate a list of median values in this array. That is, the first value to generate would be the median of all the values, the second and third values would be the two median values of the array "slice" on either side of the original median. The next four values would be the medians of the array slices around the previous two medians and so on. If you're wondering what good such a sequence could be, well, were we to store this sequence into successive elements of an array, we could develop a binary search algorithm that is a little bit faster than the algorithm above (faster by some multiplicative constant). Hence, by running this iterator over the sorted data, we can come up with a slightly faster searching algorithm.

The following is the iterator that generates this particular sequence:

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

const
    MaxData := 17;      // # of data items to process.

type
    AryType: uns32[ MaxData ];

static

    // Here is the sorted data we'll process.  For simplicity, we'll just
    // fill the array with 1..MaxData at indices 0..MaxData-1.

    SortedData: AryType :=
        [
            // Fill this table with the values 1..MaxData:

            ?i := 1;
            #while( i < MaxData )

                i,
                ?i := i + 1;

            #endwhile
            i
        ];


/*******************************************************************/
/*                                                                 */
/* MedianVal iterator-                                             */
/*                                                                 */
/* Given a sorted array, this iterator yields the median value,    */
/* then it recursively yields a list of median values for the      */
/* array slice consisting of the array elements whose index is     */
/* less than the median value.  Finally, it yields the list of     */
/* median values for the array slice built from the array elements */
/* whose indices are greater than the median element.              */
/*                                                                 */
/* Inputs:                                                         */
/*  Ary-                                                           */
/*      The array whose elements we are to process.                */
/*                                                                 */
/*  start-                                                         */
/*      Starting index for the array slice.                        */
/*                                                                 */
/*  last-                                                          */
/*      Ending index (plus one) for the array slice.               */
/*                                                                 */
/* Yields:                                                         */
/*  A list of median values in EAX (one median value on            */
/*  each iteration of the corresponding FOREACH loop).             */
/*                                                                 */
/* Notes:                                                          */
/*  This iterator wipes out EBX.                                   */
/*                                                                 */
/*******************************************************************/


iterator MedianVal( var Ary: AryType; start:uns32; last:uns32 ); nodisplay;
var
    median: uns32;
begin MedianVal;

    mov( last, eax );
    sub( start, eax );
    if( @a ) then

        shr( 1, eax );              // Compute half the size.
        add( start, eax );          // Compute median index.
        mov( eax, median );         // Save for later.
        mov( Ary, ebx );            // Compute address of median element.
        mov( [ebx][eax*4], eax );   // Get median element.
        yield();


        // Recursively yield the medians for the elements at indices below
        // the element we just yielded.  Do this by recursively calling
        // the iterator to generate the list and then yield whatever value
        // the iterator returns.

        foreach MedianVal( Ary, start, median ) do

            yield();

        endfor;

        // Recursively yield the medians for the array slice whose indices
        // are greater than the current array element.  Note that we don't
        // include the median value itself in this list as we already
        // returned that value above.

        mov( median, eax );
        inc( eax );
        foreach MedianVal( Ary, eax, last ) do

            yield();

        endfor;

    endif;

end MedianVal;


// Main program that tests the functionality of this iterator.

begin RecIter;

    foreach MedianVal( SortedData, 0, MaxData ) do

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

    endfor;

end RecIter;
```

Program 2.3 Recursive Iterator to Rearrange Data for a Binary Search

The program above produces the following output:

```
Value = 9
Value = 5
Value = 3
Value = 2
Value = 1
Value = 4
Value = 7
Value = 6
Value = 8
Value = 14
Value = 12
Value = 11
Value = 10
Value = 13
Value = 16
Value = 15
Value = 17
```

2.7 Calling Other Procedures Within an Iterator

It is perfectly legal to call other procedures from an iterator. However, unless that procedure is nested within the iterator (see “Lexical Nesting” on page 1375), you cannot yield from that procedure using the iterator’s *yield* thunk unless you pass the thunk as a parameter to the other procedure. Other than the fact that you must remember that there is additional information on the stack, calling a procedure from an iterator is really no different than calling an iterator from any other procedure.

2.8 Iterators Within Classes

You can declare an iterator within a class. Iterators are called via the class’ virtual method table, just like methods. This means that you can override an iterator at run-time. See the chapter on classes for more details.

2.9 Putting It Altogether

This chapter provides a brief introduction to the low-level implementation of iterators and then discusses several different ways that you may use iterators in your programs. Although iterators are not as familiar as other program units, they are quite useful in many important situations. You should try to use iterators in your programs wherever appropriate, even if you’re not familiar with iterators from your high level language experiences.

Coroutines and Generators Chapter Three

3.1 Chapter Overview

This chapter discusses two special types of program units known as coroutines and generators. A coroutine is similar to a procedure and a generator is similar to a function. The principle difference between these program units and procedures/functions is that the call/return mechanism is different. Coroutines are especially useful for multiplayer games and other program flow where sections of code "take turns" executing. Generators, as their name implies, are useful for generating a sequence of values; in many respects generators are quite similar to HLA's iterators without all the restrictions of the iterators (i.e., you can only use iterators within a FOREACH loop).

3.2 Coroutines

A common programming paradigm is for two sections of code to swap control of the CPU back and forth while executing. There are two ways to achieve this: preemptive and cooperative. In a preemptive system, two or more *processes* or *threads* take turns executing with the *task switch* occurring independently of the executing code. A later volume in this text will consider preemptive multitasking, where the Operating System takes responsibility for interrupting one task and transferring control to some other task. In this chapter we'll take a look at coroutines that explicitly transfer control to another section of the code[1]

When discussing coroutines, it is instructive to review how HLA's iterators work, since there is a strong correspondence between iterators and coroutines. Like iterators, there are four types of entries and returns associated with a coroutine:

- Initial entry. During the initial entry, the coroutine's caller sets up the stack and otherwise initializes the coroutine.
- Cocall to another coroutine / coreturn to the previous coroutine.
- Coreturn from another coroutine / cocall to the current coroutine.
- Final return from the coroutine (future calls require reinitialization).

A *cocall* operation transfers control between two coroutines. A cocall is effectively a call and a return instruction all rolled into one operation. From the point of view of the process executing the cocall, the cocall operation is equivalent to a procedure call; from the point of view of the processing being called, the cocall operation is equivalent to a return operation. When the second process cocalls the first, control resumes *not at the beginning of the first process*, but immediately after the last cocall operation from that coroutine (this is similar to returning from a FOREACH loop after a *yield* operation). If two processes execute a sequence of mutual cocalls, control will transfer between the two processes in the following fashion:

1. The term "cooperative" in this chapter doesn't not imply the use of that oxymoronic term "cooperative multitasking" that Microsoft and Apple used before they got their operating system acts together. Cooperative in this chapter means that two blocks of code explicitly pass control between one another. In a multiprogramming system (the proper technical term for "cooperative multitasking" the operating system still decides which program unit executes after some other thread of execution voluntarily gives up the CPU.

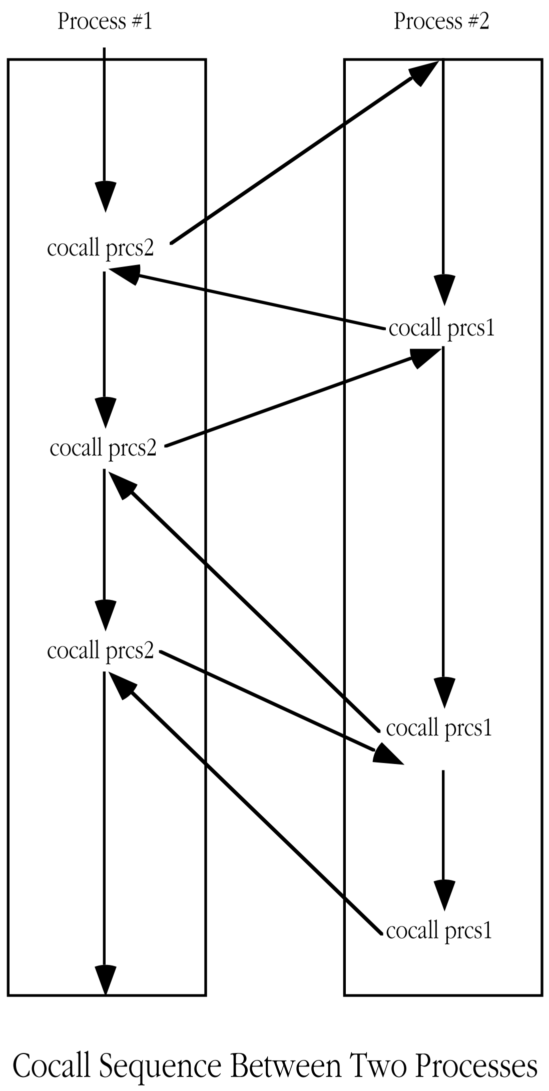

Figure 3.1 Cocall Sequence

Cocalls are quite useful for games where the "players" take turns, following different strategies. The first player executes some code to make its first move, then cocalls the second player and allows it to make a move. After the second player makes its move, it cocalls the first process and gives the first player its second move, picking up immediately after its cocall. This transfer of control bounces back and forth until one player wins.

Note, by the way, that a program may contain more than two coroutines. If coroutine one cocalls coroutine two, and coroutine two cocalls coroutine three, and then coroutine three cocalls coroutine one, coroutine one picks up immediately in coroutine one after the cocall it made to coroutine two.

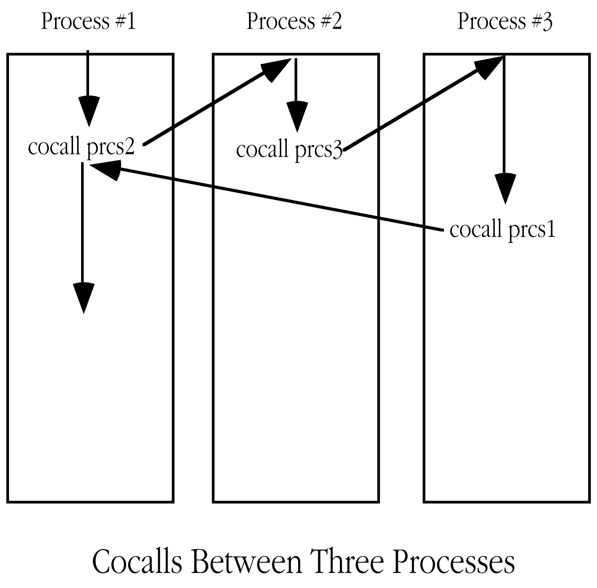

Figure 3.2 Cocalls Between Three Processes

Since a cocall effectively *returns* to the target coroutine, you might wonder what happens on the *first* cocall to any process. After all, if that process has not executed any code, there is no “return address” where you can resume execution. This is an easy problem to solve, we need only initialize the return address of such a process to the address of the first instruction to execute in that process.

A similar problem exists for the stack. When a program begins execution, the main program (coroutine one) takes control and uses the stack associated with the entire program. Since each process must have its own stack, where do the other coroutines get their stacks? There is also the question of “how much space should one reserve for each stack?” This, of course, varies with the application. If you have a simple application that doesn’t use recursion or allocate any local variables on the stack, you could get by with as little as 256 bytes of stack space for a coroutine. On the other hand, if you have recursive routines or allocate storage on the stack, you will need considerably more space. But this is getting a little ahead of ourselves, how do we create and call coroutines in the first place?

HLA does not provide a special syntax for coroutines. Instead, the HLA Standard Library provides a class with set of procedures and methods (in the coroutines library module) that lets you turn any procedure (or set of procedures) into a coroutine. The name of this class is *coroutine* and whenever you want to create a coroutine object, you need to declare a variable of type *coroutine* to maintain important state information about that coroutine’s thread of execution. Here are a couple of typical declarations that might appear within the VAR section of your main program:

```
var
    FirstPlayer: pointer to coroutine;
    OtherPlayer: coroutine;
```

Note that a coroutine variable is not the coroutine itself. Instead, the coroutine variable keeps track of the machine state when you switch between the declared coroutine and some other coroutine in the program (including the main program, which is a special case of a coroutine). The coroutine’s "body" is a procedure that you write independently of the coroutine variable and associate with that coroutine object.

The coroutine class contains a constructor that uses the conventional name *coroutine.create*. This constructor requires two parameters and has the following prototype:

```
    procedure coroutine.create( stacksize:dword;  body:procedure );
```

The first parameter specifies the size (in bytes) of the stack to allocate for this coroutine. The construction will allocate storage for the new coroutine in the system's heap using dynamic allocation (i.e., *malloc*). As a general rule, you should allocate at least 256 bytes of storage for the stack, more if your coroutine requires it (for local variables and return addressees). Remember, the system allocates all local variables in the coroutine (and in the procedures that the coroutine calls) on this stack; so you need to reserve sufficient space to accommodate these needs. Also note that if there are any recursive procedures in the coroutine's thread of execution, you will need some additional stack space to handle the recursive calls.

The second parameter is the address of the procedure where execution begins on the first cocall to this coroutine. Execution begins with the first executable statement of this procedure. If the procedure has some local variables, the procedure must build a stack frame (i.e., you shouldn't specify the NOFRAME procedure option). Procedures you execute via a cocall should never have any parameters (the calling code will not properly set up those parameters and references to the parameter list may crash the machine).

This constructor is a conventional HLA class constructor. On entry, if ESI contains a non-null value, the constructor assumes that it points at a coroutine class object and the constructor initializes that object. On the other hand, if ESI contains NULL upon entry into the constructor, then the constructor allocates new storage for a coroutine object on the heap, initializes that object, and returns a pointer to the object in the ESI register. The examples in this chapter will always assume dynamic allocation of the coroutine object (i.e., we'll use pointers).

To transfer control from one coroutine (including the main program) to another, you use the *coroutine.cocall* method. This method, which has no parameters, switches the thread of execution from the current coroutine to the coroutine object associated with the method. For example, if you're in the main program, the following two cocalls transfer control to the *FirstPlayer* and then the *OtherPlayer* coroutines:

```
FirstPlayer.cocall();
OtherPlayer.cocall();
```

There are two important things to note here. First, the syntax is not quite the same as a procedure call. You don't use cocall along with some operand that specifies which coroutine to transfer to (as you would if this were a CALL instruction); instead, you specify the coroutine object and invoke the cocall method for that object. The second thing to keep in mind is that these are coroutine transfers, not subroutine calls; therefore, the *FirstPlayer* coroutine doesn't necessarily return back to this code sequence. *FirstPlayer* could transfer control directly to *OtherPlayer* or some other coroutine once it finishes whatever tasks it's working on. Some coroutine has to explicitly transfer control to the thread of execution above in order for it to (directly) transfer control to *OtherPlayer.*

Although coroutines are more general than procedures and don't have to use the call/return semantics, it is perfectly possible to simulate and call and return exchange between two coroutines. If one coroutine calls another and that second coroutine cocalls the first, you get semantics that are very similar to a call/return (of course, on the next call to the second coroutine control resumes after the cocall, not at the start of the coroutine, but we will ignore that difference here). The only problem with this approach is that it is not general: both coroutines have to be aware of the other. Consider a cooperating pair of coroutines, *master* and *slave*. The *master* coroutine corresponds to the main program and the *slave* coroutine corresponds to a procedure that the main program calls. Unfortunately, *slave* is not general purpose like a standard procedure because it explicitly calls the *master* coroutine (at least, using the techniques we've seen thus far). Therefore, you cannot call it from an arbitrary coroutine and expect control to transfer back to that coroutine. If *slave* contains a cocall to the *master* coroutine it will transfer control there rather than back to the "calling" coroutine. Although these are the semantics we expect of coroutines, it would sometimes be nice if a coroutine could return to whomever invoked it without explicitly knowing who invoked it. While it is possible to set up some coroutine variables and pass this information between coroutines, the HLA Standard Library Coroutines Module provides a better solution: the *coret* procedure.

On each call to a coroutine, the coroutine run-time support code remembers the last coroutine that made a cocall. The *coret* procedure uses this information to transfer control back to the last coroutine that made a cocall. Therefore, the *slave* coroutine above can execute the coret procedure to transfer control back to whomever called it without knowing who that was.

Note that the *coret* procedure is not a member of the *coroutine* class. Therefore, you do not preface the call with "coroutine." You invoke it directly:

```
coret();
```

Another important issue to keep in mind with *coret* is that it only keeps track of the last cocall. It does not maintain a stack of coroutine "return addresses" (there are several different stacks in use by coroutines, on which one does it keep this information?). Therefore, you cannot make two cocalls in a row and then execute two corets to return control back to the original coroutine. If you need this facility, you're going to need to create and maintain your own stack of coroutine calls. Fortunately, the need for something like this is fairly rare. You generally don't use coroutines as though they were procedures and in the few instances where this is convenient, a single level of return address is usually sufficient.

By default, every HLA main program is a coroutine. Whenever you compile an HLA program (as opposed to a UNIT), the HLA compiler automatically inserts two pieces of extra code at the beginning of the main program. The first piece of extra code initializes the HLA exception handling system, the second sets up a coroutine variable for the main program. The purpose of this variable is to all other coroutines to transfer control back to the main program; after all, if you transfer control from one coroutine to another using a statement like *VarName.cocall*, you're going to need a coroutine variable associated with the main program in order to cocall the main program. HLA automatically creates and initializes this variable when execution of the main program begins. So the only question is, "how do you gain access to this variable?"

The answer is simply, really. Whenever you include the "coroutines.hhf" header file (or "stdlib.hhf" which automatically includes "coroutines.hhf") HLA declares a static coroutine variable for you that is associated with the main program's coroutine object. That declaration looks something like the following[2]:

```
static MainPgm:coroutine; external( "<<external name for MainPgm>>" );
```

Therefore, to transfer control from one coroutine to the main program's coroutine, you'd use a cocall like the following:

```
MainPgm.cocall();
```

The last method of interest to us in the *coroutine* class is the *coroutine.cofree* method. This is the destructor for the *coroutine* class. Calling this method frees up the stack storage associated with the coroutine and cleans up other state information associated with that coroutine. A typical call might look like the following:

```
    OtherPlayer.cofree();
```

Warning: do not call the *cofree* method from within the coroutine you are freeing up. There is no guarantee that the stack and coroutine state variables remain valid for a given coroutine after you call *cofree*. Generally, it is a good idea to call the *cofree* method in the same code that originally created the coroutine via the *coroutine.create* call. It goes without saying that you must not call a coroutine after you've destroyed it via the *cofree* method call.

Remember that the *coret* procedure call is really a special form of *cocall*. Therefore, you need to be careful about executing *coret* after calling the *cofree* method as you may wind up "returning" to the coroutine you just destroyed.

After you call the *cofree* method, it is perfectly reasonable create a new coroutine using that same coroutine variable by once again calling the *coroutine.create* procedure. However, you should always ensure that you call the *cofree* method prior to calling *coroutine.create* or the stack space allocated in the original call will be lost in the system (i.e., you'll create a memory leak).

This is very important: you never "return" from a coroutine using a RET instruction (e.g., by "falling off the end of the procedure) or via the HLA EXIT/EXITIF statement. The only legal ways to "return" from a coroutine are via the *cocall* and *coret* operations. If you RETurn or EXIT from a coroutine, that coroutine enters a special mode that rejects any future cocalls and immediately returns control back to whomever

2. The external name doesn't appear here because it is subject to change. See the coroutines.hhf header file if you need to know the actual external name for some reason.

cocalled it in the first place. Most coroutines contain an infinite loop that transfers control back to the start of the coroutine to repeat whatever function they perform once they complete all the code in the coroutine. You will probably want to implement this functionality in your coroutines as well.

3.3 Parameters and Register Values in Coroutine Calls

As you've probably noticed, coroutine calls via cocall don't support generic parameters for transferring data between two coroutines. There are a couple of reasons for this. First of all, passing parameters to a coroutine is difficult because we typically use the stack to pass parameters and coroutines all use different stacks. Therefore, the parameters one coroutine passes to another won't be on the correct stack (and, therefore, inaccessible) when the second coroutine continues execution. Another problem with passing parameters between coroutines is that a typical coroutine has several entry points (immediately after each cocall in that coroutine). Nevertheless, it is often important to communicate information between coroutines. We will explore ways to do that in this section.

If we can pass parameters on the stack, that basically leaves registers and global memory locations[3]. Registers are the easy and obvious solution. However, keep in mind that you have a limited set of registers available so you can't pass much data between coroutines in the registers. On the other hand, you can pass a pointer to a block of data in a register (see "Passing Parameters via a Parameter Block" on page 1353 for details).

Another place to pass parameters between coroutines is in global memory locations. Keep in mind that coroutines each have their own stack. Therefore, one coroutine does not have access to another coroutine's automatic variables appearing in a VAR section (this is true even if those VAR objects appear in the main program). Always use STATIC, STORAGE, or READONLY objects when communicating data between coroutines using global variables. If you must communicate an automatic or dynamic object, pass the address of that object in a register or some static global variable.

A bigger problem than where we pass parameters to a coroutine is "How do we deal with parameters we pass to a coroutine?" Parameters work well in procedures because we always enter the procedure at the same point and the state of the procedure is usually the same upon entry (with the possible exception of static variable values). This is not true for coroutines. Consider the following code that executes as a coroutine:

```
procedure IsACoroutine; nodisplay; noframe;
begin IsACoroutine;

    //*

    << Do something upon initial entry >>

    coret();    // Invoke previous coroutine
    //*

    << Do some more stuff upon return >>

    forever

        OtherCoroutine.cocall();  // Invoke a third coroutine.

        << Do some more stuff here >>

        MainPgm.cocall();         // Transfer control back to the main program.
        //*

        << do some stuff >>
```

3. The chapter on low-level parameter implementation in this volume discusses different places you can pass parameters between procedures. Check out that chapter for more details on this subject.

```
        coret();                 // Return to whomever cocalled us.
        //*

        << do some more stuff >>

    endfor;

end IsACoroutine;
```

In this code you'll find several comments of the form "//*". These comments mark the point at which some other coroutine can reenter this code. Note that, in general, the "calling" coroutine has no idea which entry point it will invoke and, likewise, this coroutine has no idea who invoked it at any given point. Passing in meaningful parameters and properly processing them under these conditions is difficult, at best. The only reasonable solution is to make every invocation pass exactly the same type of data in the same location and then write your coroutines to handle this data appropriately upon each entry. Even though this solution is more reasonable than the other possibilities, maintaining code like this is very difficult.

If you're really dead set on passing parameters to a coroutine, the best solution is to have a single entry point into the code so you've only got the handle the parameter data in one spot. Consider the following procedure that other threads invoke as a coroutine:

```
procedure HasAParm; nodisplay;
begin HasAParm;

    << Initialization code goes here, assume no parameter >>
    forever

        coret();  // Or cocall some other coroutine.

        << deal with parameter data passed into this coroutine >>

    endfor;

end HasAParm;
```

Note that there are two entry points into this code: the first occurs on the initial entry. The other occurs whenever the *coret*() procedure returns via a *cocall* to this coroutine. Immediately after the *coret* statement, the code above can process whatever parameter data the calling code has set up. After processing that data, this code returns to the invoking coroutine and that coroutine (directly or indirectly) can invoke this code again with more data.

3.4 Recursion, Reentrancy, and Variables

The fact that each coroutine has its own stack impacts access to variables from within coroutines. In particular, the only automatic (VAR) objects you can normally access are those declared within the coroutine itself and any procedures it calls. In a later chapter of this volume we'll take a look at nested procedures and how one could access local variables outside of the current procedure. For the most part, that discussion does not apply to procedures that are coroutines.

If you wish to share information between two or more coroutines, the best place to put such information is in a static object (STATIC, READONLY, and STORAGE variables). Such data does not appear on a stack and is accessible to multiple coroutines (and other procedures) simultaneously.

Whenever you execute a cocall instruction, the system suspends the current thread of execution and switches to a different coroutine, effectively by returning to that other coroutine and picking up where it left off (via a *coret* or *cocall* operation). This means that it isn't really possible to recursively *cocall* some coroutine. Consider the following (vain) attempt to achieve this:

```
procedure IsACoroutine; nodisplay;
```

```
begin IsACoroutine;

    << Do some initial stuff >>

    IAC.cocall();  // Note: IAC is initialized with the address of IsACoroutine.

    << Do some more stuff >>

end IsACoroutine;
```

This code assumes that *IAC* is a coroutine variable initialized with the address of the *IsACoroutine* procedure. The *IAC.cocall* statement, therefore, is an attempt to recursively call this coroutine. However, all that happens is that this call leaves the *IsACoroutine* code and then the coroutine system passes control to where *IAC* last left off. That just happens to be the *IAC.cocall* statement that just left the *IsACoroutine* procedure. Therefore this code immediately returns back to itself.

Although the idea of a recursive coroutine doesn't make sense, it is certainly possible to call procedures from within a coroutine and those procedures can be recursive. You can even make cocalls from those (recursive) procedures and the coroutine system will automatically return back into the recursive calls when control transfers back to the coroutine.

Although coroutines are not recursive, it is quite possible for the coroutine run-time system to reenter a procedure that is a coroutine. This situation can occur when you have two coroutine variables, initialize them both with the address of the same procedure, and then execute a cocall to each of the coroutine objects. consider the following simple example:

```
procedure Reentered; nodisplay;
begin Reentered;

    << do some initialization or other work >>

    coret();

    << do some other stuff >>

end Reentered;
        .
        .
        .
    CV1.create( 256, &Reentered );  // Initialize two coroutine variables with
    CV2.create( 256, &Reentered );  // the address of the same procedure.
    CV1.cocall();                   // Start the first coroutine.
    CV2.cocall();                   // Start the second coroutine.

    << At this point, both CV1 and CV2 are suspended within Reentered >>
```

Notice at the end of this code sequence, both the CV1 and CV2 coroutines are executing inside the Reentered procedure. That is, if you cocall either one of them, the cocalled routine will continue execution after the *coret* statement. This is not an example of a recursive coroutine call, but it certainly demonstrates that you can reenter some procedure while some other coroutine is currently executing the code within that procedure.

Reentering some code (whether you do this via a coroutine call or some other mechanism) is a perfectly reasonable thing to do. Indeed, recursion is one example of reentrancy. However, there are some special considerations you must be aware of when it is possible to reenter some code.

The principal issue is the use of variables in reentrant code. Suppose the *Reentered* procedure above had the following declarations:

```
    var
        i: int32;
        j: uns32;
```

One concern you might have is that the two different coroutines executing in the same procedure would share these variables. However, keep in mind that HLA allocates automatic variables on the stack. Since each coroutine has its own stack, they're going to get their own private copies of the variables. Therefore, if *CV1* stores a value into *i* and *j*, *CV2* will not see these values. While this may seem to be a problem, this is actually what you want. You generally don't want one coroutine's thread of execution affecting the calculation in a different thread of execution.

The discussion above applies only to automatic variables. HLA does not allocate static objects (those you declare in the STATIC, READONLY, and STORAGE sections) on the stack. Therefore, such variables are not associated with a specific coroutine; instead, all coroutines that are executing in the same procedure share the same variables. Therefore, you shouldn't use static variables within a procedure that serves as a coroutine (especially reentrant coroutines) unless you explicitly want to share that data among other coroutines or other procedures.

Coroutines have their own stack and maintain that stack between cocalls to other coroutines. Therefore, like iterators, coroutines maintain their state (including the value of automatic variables) across cocalls. This is true even if you leave a coroutine via some other procedure than the main procedure for the coroutine. For example, suppose coroutine *A* calls some procedure *B* and then within procedure *B* there is a cocall to some other coroutine *C*. Whenever coroutine *C* executes *coret* or some coroutine (including *C*) cocalls *A*, control transfers back into procedure *B* and procedure *B's* state is maintained (including the values of all local variables *B* initialize prior to the cocall). When *B* executes a return instruction, it will return back to procedure *A* who originally called *B*.

In theory it's even possible to call a procedure as well as cocall that procedure (it's hard to imagine why you would want to do this and it's probably quite difficult to pull it off correctly, but it's certainly possible). This is such a bizarre situation that we won't consider it any farther here.

3.5 Generators

A generator is to a function what a coroutine is to a procedure. That is, the whole purpose of a generator is to return a value like a function result. As far as HLA and the Coroutines Library Module is concerned, there is absolutely no difference between a generator and a coroutine (anymore than there is a syntactical difference between a function and a procedure to HLA). Clearly, there are some semantic differences; this section will describe the semantics of a generator and propose a convention for generator implementation.

The best way to describe a generator is to begin with the discussion of a special-purpose generator object that HLA does support – the iterator. An iterator is a special form of a generator that does not require its own stack. Iterators share the same stack as the calling code (i.e., the code containing the FOREACH loop that invokes the iterator). Because of the semantics of the FOREACH loop, iterators can leave their activation records on the stack and, therefore, maintain their local state between exits to the FOREACH loop body. The disadvantage to this scheme is that the calling semantics of an iterator are very rigidly defined; you cannot call an iterator from an arbitrary point in a program and the iterator's state is preserved only for the execution of the FOREACH loop.

By using its own stack, a generator removes these restrictions. You can call a generator from any point in the program (except, of course, within the generator itself – remember, recursive coroutines are not possible). Also, the state (i.e., the activation record) of a generator is not tied to the execution of some syntactical item like a FOREACH loop. The generator maintains its local state from the point of its first call to the point you call *cofree* on that generator.

One major difference between iterators and generators is the fact that generators don't use the *yield* statement (thunk) to return results back to the calling code. To send a value back to whomever invokes the generator, the generator must cocall the original coroutine. Since one can call a generator from different points in the code, and in particular, from different coroutines, the typical way to "return" a value back to the "caller" is to use the *coret* procedure call after loading the return result into a register.

A typical generator does not use the *cocall* operation. The *cocall* method transfers control to some other (explicitly defined) coroutine. As a general rule, generators (like functions) return control to whomever called them. They do not explicitly pass control through to some other coroutine. Keep in mind that the *coret* procedure only returns control to the last coroutine. Therefore, if some generator passes control to another coroutine, there is no way to anonymously return back to whomever called the generator in the first place. That information is lost at the point of the second *cocall* operator. Since the main purpose of a generator is to return a value to whomever cocalled it, finding cocalls in a generator would be unusual indeed.

One problem with generators is that, like the coroutines upon which they are based, you cannot pass parameters to a generator via the stack. In most cases this is perfectly acceptable. As their name implies, generators typically generate an independent stream of data once they begin execution. After initialization, a generator generally doesn't require any additional information in order to generate its data sequence. Although this is the typical case, it is not the only case; sometimes you may need to write generators that need parameter data on each call. So what's the best way to handle this?

In a previous section (see "Parameters and Register Values in Coroutine Calls" on page 1334) we discussed a couple of ways to pass parameters to coroutines. While those techniques apply here as well, they are not particularly convenient (certainly not as convenient as passing parameters to a standard HLA procedure). Because of their function-like nature, it is more common to have to pass parameters to a generator (versus a generic coroutine) and you'll probably make more calls to generators that require parameters (versus similar calls to coroutines). Therefore, it would be nice if there were a "high-level" way of passing parameters to generators. Well, with a couple of tricks we can easily accomplish this[4].

Remember that we cannot pass parameters to a coroutine (or generator) on the stack. The most convenient place to pass coroutine parameters is in registers or in static, global, memory locations. Unfortunately, writing a sequence of instructions to load up registers with parameter values (or, worse yet, copy the parameter data to global variables) prior to invoking a generator is a real pain. Fortunately, HLA's macros come to the rescue here; we can easily write a macro that lets us invoke a generator using a high level syntax and the macro can take care of the dirty work of loading registers (or global memory locations) with the necessary values. As an example, consider a generator, *_MyGen*, that expects two parameters in the EAX and EBX registers. Here's a macro, *MyGen*, that sets up the registers and invokes this generator:

```
#macro MyGen( ParmForEAX, ParmForEBX );

    mov( ParmForEAX, eax );
    mov( ParmForEBX, ebx );
    _MyGen.cocall();

#endmacro;
    .
    .
    .
MyGen( 5, i );
```

You could, with just a tiny bit more effort, pass the parameters in global memory locations using this same technique.

In a few situations, you'll really need to pass parameters to a generator on the stack. We'll not go into the reasons or details here, but there are some rare circumstances where this is necessary. In many other circumstances, it may not be necessary but it's certainly more convenient to pass the parameters on the stack because you get to take advantage of HLA's high level parameter passing syntax when you use this scheme (i.e., you get to choose the parameter passing mechanism and HLA will automatically handle a lot of the gory details behind parameter passing for you when you use the stack). The best solution in this situation is to write a wrapper procedure. A wrapper procedure is a short procedure that reorganizes a parameter list before calling some other procedure. The macro above is a simple example of a wrapper – it takes two the (text) parameters and moves their run-time data into EAX and EBX prior to cocalling *_MyGen*. We could have just as easily written a procedure to accomplish this same task:

```
procedure MyGen( ParmForEAX:dword; ParmForEBX:dword ); nodisplay;
begin MyGen;
```

4. By the way, generators are coroutines, so these tricks apply to generic coroutines as well.

```
    mov( ParmForEAX, eax );
    mov( ParmForEBX, ebx );
    _MyGen.cocall();

end MyGen;
```

Beyond the obvious time/space trade-offs between macros and procedures, there is one other big difference between these two schemes: the procedure variation allows you to specify a parameter passing mechanism like pass by reference, pass by value/result, pass by name, etc[5]. Once HLA knows the parameter passing mechanism, it can automatically emit code to process the actual parameters for you. Suppose, for example, you needed pass by value/result semantics. Using the macro invocation, you'd have to explicitly write a lot of code to pull this off. In the procedure above, about the only change you'd need is to add some code to store any returned results back into *ParmForEAX* or *ParmForEBX* (whichever uses the pass by value/result mechanism).

Since coroutines and generators share the same memory address space as the calling coroutine, it is not correct to say that a coroutine or generator does not have access to the stack of the calling code. The stack is in the same address space as the coroutine/generator; the only problem is that the coroutine doesn't know exactly where any parameters may be sitting in that memory space. This is because procedures use the value in ESP[6] to indirectly reference the parameters passed on the stack and, unfortunately, the *cocall* method changes the value of the ESP register upon entry into the coroutine/generator. However, were we to pass the original value of ESP (or some other pointer into an activation record) through to a generator, then it would have direct access to those values on the stack. Consider the following modification to the *MyGen* procedure above:

```
procedure MyGen( ParmForEAX:dword; ParmForEBX:dword ); nodisplay;
begin MyGen;

    mov( ebp, ebx );
    _MyGen.cocall();

end MyGen;
```

Notice that this code does not directly copy the two parameters into some locations that are directly accessible in the generator. Instead, this procedure simply copies the base address of *MyGen's* activation record (the value in EBP) into the EBX register for use in the generator. To gain access to those parameters, the generator need only index off of EBX using appropriate offsets for the two parameters in the activation record. Perhaps the easiest way to do this is by declaring an explicit record declaration that corresponds to *MyGen's* activation record:

```
type
    MyGenAR:
        record
            OldEBP:     dword;
            RtnAdrs:    dword;
            ParmForEAX: dword;
            ParmForEBX: dword;
        endrecord;
```

Now, within the *_MyGen* generator code, you can access the parameters on the stack using code like the following:

```
    mov( (type MyGenAR [ebx]).ParmForEAX, eax );
    mov( (type MyGenAR [ebx]).ParmForEBX, edx );
```

5. For a discussion of pass by value/result and pass by name parameter passing mechanisms, see the chapter on low-level parameter implementation in this volume.
6. Okay, a procedure typically uses the value in EBP, but that same procedure also loads EBP with the value of ESP in the standard entry sequence.

This scheme works great for pass by value and pass by reference parameters (those we've seen up to this point). There are other parameter passing mechanisms, this scheme doesn't work as well for those other parameter passing mechanisms. Fortunately, you won't use those other parameter passing methods anywhere near as often as pass by value and pass by name.

3.6 Exceptions and Coroutines

Exceptions represent a special problem in coroutines. The HLA TRY..ENDTRY statement typically surrounds a block of statements one wishes to protect from errant code. The TRY.. ENDTRY statement is a dynamic control structure insofar as this statement also protects any procedures you call from within the TRY..ENDTRY block. So the obvious question is "does the TRY..ENDTRY statement protect a coroutine you call from within such a block as well?" The short answer is "no, it does not."

Actually, in the first implementation of the HLA coroutines module, the exception handling system did pass control from one coroutine to another whenever an exception occurred. However, it became immediately obvious that this behavior causes non-intuitive results. Coroutines tend to be independent entities and to have one coroutine transfer control to another without an explicit cocall creates some problems. Therefore, the HLA compiler and the coroutines module now treat each coroutine as a standalone entity as far as exceptions are concerned. If an exception occurs within some coroutine and there isn't an outstanding TRY..ENDTRY block active for that coroutine, then the system behaves as though there were no active TRY..ENDTRY at all, *even if there is an active TRY..ENDTRY block in another coroutine*; in other words, the program aborts execution. Keep this in mind when using exception handling in your coroutines. For more details on exception handling, see the chapter on Exception Handling in this volume.

3.7 Putting It All Together

This chapter discusses a novel program unit – the coroutine. Coroutines have many special properties that make them especially valuable in certain situations. Coroutines are not built into the HLA language. Rather, HLA implements them via the HLA Coroutines Module in the HLA Standard Library. This chapter began by discussing the methods found in that library module. Next, this chapter discusses the use of variables, recursion, reentrancy and machine state in a coroutine. This chapter also discusses how to create generators using coroutines and pass parameters to a generator in a function-like fashion. Finally, this chapter briefly discussed the use of the TRY..ENDTRY statement in coroutines.

Coroutines and generators are like iterators insofar as they are control structures that few high level languages implement. Therefore, most programmers are unfamiliar with the concept of a coroutine. This, unfortunately, leads to the lack of consideration of coroutines in a program, even where a coroutine is the most suitable control structure to use. You should avoid this trap and learn how to use coroutines and generators properly so that you'll know when to use them when the need arises.

Advanced Parameter Implementation Chapter Four

4.1 Chapter Overview

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

4.2 Parameters

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

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

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

4.3 Where You Can Pass Parameters

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

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

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

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

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

4.3.1 Passing Parameters in (Integer) Registers

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

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

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

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

First Last

eax, edx, esi, edi, ebx, ecx

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

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

```
varname : typename in register
```

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

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

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

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

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

        or( $20, chToSwap );

    endif;
    ret();

end swapcase;
```

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

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

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

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

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

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

        or( $20, al );

    endif;
    ret();

end swapcase;
```

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

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

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

// swapcase( charVar );

    mov( charVar, al );
    call swapcase;

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

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

// swapcase( ah );

    mov( ah, al );
    call swapcase;

// swapcase( al );

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

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

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

```
// memfill- This procedure stores <ECX> copies of the byte in AL starting
// at the memory location specified by EDI:

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

begin memfill;

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

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

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

end memfill;
```

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

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

begin memfill;

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

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

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

end memfill;
```

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

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

4.3.2 Passing Parameters in FPU and MMX Registers

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

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

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

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

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

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

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

Keep in mind the limited size of the FPU stack. This effectively eliminates the possibility of passing real parameter values through the FPU registers in a recursive procedure. Also keep in mind that it is rather difficult to preserve FPU register values across a procedure call, so be careful about using the FPU registers to pass parameters since such operations could disturb the values already on the FPU stack (e.g., cause an FPU stack overflow).

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

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

4.3.3 Passing Parameters in Global Variables

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

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

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

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

end ThisProc;
          .
          .
          .

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

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

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

4.3.4 Passing Parameters on the Stack

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

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

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

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

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

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

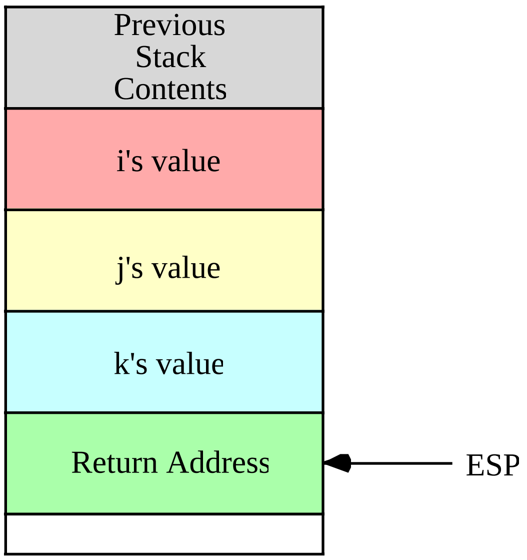

Figure 4.1 Activation Record for CallProc Invocation

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

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

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

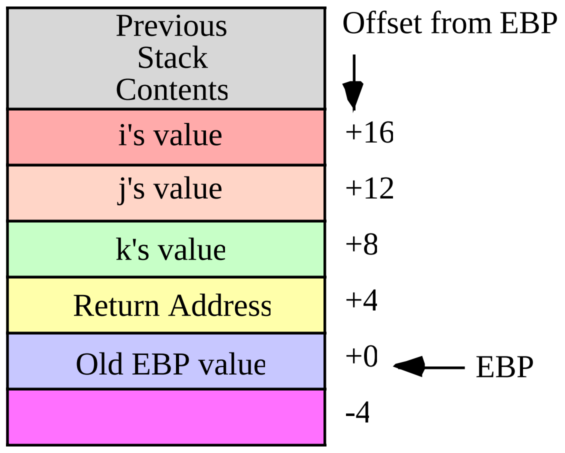

Figure 4.2 Offsets into CallProc's Activation Record

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

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

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

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

Although pushing the actual parameters on the stack in the order of the formal parameters' declarations is very common (and the default case that HLA uses), this is not the only order a program can use. Some high level languages (most notably, C, C++, Java, and other C-derived languages) push their parameters in the reverse order, that is, from right to left. The primary reason they do this is to allow variable parameter lists, a subject we will discuss a little later in this chapter (see “Variable Parameter Lists” on page 1368). Because it is very common for programmers to interface HLA programs with programs written in C, C++, Java, and other such languages, HLA provides a mechanism that allows it to process parameters in this order.

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

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

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

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

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

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

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

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

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

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

Another issue concerning the use of parameters on the stack is "who takes the responsibility for cleaning these parameters off the stack?" As you saw in the chapter on Mixed Language Programming, various languages assign this responsibility differently. For example, in languages like Pascal, it is the procedure’s responsibility to clean up parameters off the stack before returning control to the caller. In languages like C/C++, it is the caller’s responsibility to clean up parameters on the stack after the procedure returns. By default, HLA procedures use the Pascal calling convention, and therefore the procedures themselves take responsibility for cleaning up the stack. However, if you specify the @CDECL procedure option for a given procedure, then HLA does not emit the code to remove the parameters from the stack when a procedure returns. Instead, HLA leaves it up to the caller to remove those parameters. Therefore, the call above to CallProc (the one with the @CDECL option) isn't completely correct. Immediately after the call the code should remove the 12 bytes of parameters it has pushed on the stack. It could accomplish this using code like the following:

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

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

```
pushd( 5 );
call Proc1Parm

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

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

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

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

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

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

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

4.3.5 Passing Parameters in the Code Stream

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

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

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

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

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

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

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

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

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

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

One problem with passing parameters in the code stream is that the code stream is read-only[2]. Therefore, any parameter you pass in the code stream must, necessarily, be a constant. While one can easily dream up some functions to whom you always pass constant values in the parameter lists, most procedures work best if you can pass different values (through variables) on each call to a procedure. Unfortunately,

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

this is not possible when passing parameters by value to a procedure through the code stream. Fortunately, we can also pass data by reference through the code stream.

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

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

Calling sequence:

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

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

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

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

end AddEm;
```

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

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

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

4.3.6 Passing Parameters via a Parameter Block

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

Calling sequence:

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

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

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

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

end AddEm;
```

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

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

Parameter blocks are especially useful when you make several different calls to a procedure and in each instance you pass constant values. Parameter blocks are less useful when you pass variables to procedures, because you will need to copy the current variable's value into the parameter block before the call (this is roughly equivalent to passing the parameter in a global variable. However, if each particular call to a procedure has a fixed parameter list, and that parameter list contains constants (static addresses or constant values), then using parameter blocks can be a useful mechanism.

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

4.4 How You Can Pass Parameters

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

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

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

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

4.4.1 Pass by Value-Result

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

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

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

HLA supports the use of value/result parameters via the VALRES keyword. If you prefix a parameter declaration with VALRES, HLA will assume you want to pass the parameter by value/result. Whenever you call the procedure, HLA treats the parameter like a pass by reference parameter and generates code to pass the address of the actual parameter to the procedure. Within the procedure, HLA emits code to copy the data referenced by this point to a local copy of the variable[4]. In the body of the procedure, you access the param-

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

eter as though it were a pass by value parameter. Finally, before the procedure returns, HLA emits code to copy the local data back to the actual parameter. Here's the syntax for a typical procedure that uses pass by value result:

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

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

end AddandZero;
```

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

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

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

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

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

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

    // Actual procedure body begins here:

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

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

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

end AddandZero;
```

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

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

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

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

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

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

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

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

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

    mov( i, ebx );
    mov( 4, (type int32 [ebx]) );
    mov( j, ecx );
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

end DisplayAndClear;
    .
```

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

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

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

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

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

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

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

4.4.2 Pass by Result

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

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

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

```
    mov( 5, r );

end HasResParm;
```

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

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

4.4.3 Pass by Name

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

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

    foreach index in 0..10 do

        item := 0;

    endfor;

end PassByName;
```

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

```
begin PassByName;

    foreach I in 0..10 do

        A[I] := 0;

    endfor;

end PassByName;
```

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

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

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

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

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

end ItemThunk;
```

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

```
; item := 0;

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

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

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

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

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

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

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

        endwhile;

    end passByName;
```

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

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

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

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

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

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

4.4.4 Pass by Lazy-Evaluation

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

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

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

Consider the *PassByEval* procedure above. Suppose it takes several minutes to compute the values for the *a*, *b*, and *c* parameters (these could be, for example, three different possible paths in a Chess game). Perhaps the *PassByEval* procedure only uses the value of one of these parameters. Without pass by lazy evaluation, the calling code would have to spend the time to compute all three parameters even though the procedure will only use one of the values. With pass by lazy evaluation, however, the procedure will only spend the time computing the value of the one parameter it needs. Lazy evaluation is a common technique artificial intelligence (AI) and operating systems use to improve performance since it provides deferred parameter evaluation capability.

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

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

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

4.5 Passing Parameters as Parameters to Another Procedure

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

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

4.5.1 Passing Reference Parameters to Other Procedures

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

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

procedure HasRef(var refparm:dword);

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

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

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

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

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

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

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

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

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

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

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

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

```
            ToProc( refparm );
```

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

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

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

4.5.2 Passing Value-Result and Result Parameters as Parameters

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

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

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

4.5.3 Passing Name Parameters to Other Procedures

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

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

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

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

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

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

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

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

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

4.5.4 Passing Lazy Evaluation Parameters as Parameters

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

4.5.5 Parameter Passing Summary

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

Table 1: Passing Parameters as Parameters to Another Procedure

	Pass as Value	Pass as Reference	Pass as Value-Resu lt	Pass as Result	Pass as Name	Pass as Lazy Evaluation
Value	Pass the value	Pass address of the value parameter	Pass address of the value parameter	Pass address of the value parameter	Create a thunk that returns the address of the value parameter	Create a thunk that returns the value

Table 1: Passing Parameters as Parameters to Another Procedure

	Pass as Value	Pass as Reference	Pass as Value-Result	Pass as Result	Pass as Name	Pass as Lazy Evaluation
Reference	Dereference parameter and pass the value it points at	Pass the address (value of the reference parameter)	Pass the address (value of the reference parameter)	Pass the address (value of the reference parameter)	Create a thunk that passes the address (value of the reference parameter)	Create a thunk that deferences the reference parameter and returns its value
Value-Result	Pass the local value as the value parameter	Pass the address of the local value as the parameter	Pass the address of the local value as the parameter	Pass the address of the local value as the parameter	Create a thunk that returns the address of the local value of the value-result parameter	Create a thunk that returns the value in the local value of the value-result parameter
Result	Pass the local value as the value parameter	Pass the address of the local value as the parameter	Pass the address of the local value as the parameter	Pass the address of the local value as the parameter	Create a thunk that returns the address of the local value of the result parameter	Create a thunk that returns the value in the local value of the result parameter
Name	Call the thunk, dereference the pointer, and pass the value at the address the thunk returns	Call the thunk and pass the address it returns as the parameter	Call the thunk and pass the address it returns as the parameter	Call the thunk and pass the address it returns as the parameter	Pass the address of the thunk and any other values associated with the name parameter	Write a thunk that calls the name parameter's thunk, dereferences the address it returns, and then returns the value at that address

Table 1: Passing Parameters as Parameters to Another Procedure

	Pass as Value	Pass as Reference	Pass as Value-Resu lt	Pass as Result	Pass as Name	Pass as Lazy Evaluation
Lazy Evaluation	If neces-sary, call the thunk to obtain the Lazy Eval parame-ter's value. Pass the local value as the value parameter	Not possi-ble. Lazy Eval param-eters return a value which does not have an address.	Not possi-ble. Lazy Eval param-eters return a value which does not have an address.	Not possi-ble. Lazy Eval param-eters return a value which does not have an address.	Not possi-ble. Lazy Eval param-eters return a value which does not have an address.	Create a thunk that calls the caller's Lazy Eval parameter. This new thunk returns that result as its result.

4.6 Variable Parameter Lists

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

            if( eax < [ebx] ) then

                mov( [ebx], eax );

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

        until( ecx = 0 );

    else

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

        xor( eax, eax );

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

end MaxUns32;
```

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

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

4.7 Function Results

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

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

4.7.1 Returning Function Results in a Register

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

Use	First	Last

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

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

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

4.7.2 Returning Function Results on the Stack

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

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

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

      << code that leaves a value in RtnResult >>

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

end RtnOnStack;
```

Calling sequence:

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

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

4.7.3 Returning Function Results in Memory Locations

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

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

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

4.7.4 Returning Large Function Results

Returning small scalar values in the registers or on the stack makes a lot of sense. However, mechanism for returning function results does not scale very well to large data structures. The registers are too small to return large records or arrays and returning such data on the stack is a lot of work (not to mention that you’ve got to copy the data from the stack to it’s final resting spot upon return from the function). In this section we’ll take a look at a couple of methods for returning large objects from a function.

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

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

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

4.8 Putting It All Together

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

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

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

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

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

Lexical Nesting Chapter Five

5.1 Chapter Overview

This chapter discusses nested procedures and the issues associated with calling such procedure and accessing local variables in nested procedures. Nesting procedures offers HLA users a modicum of built-in information hiding support. Therefore, the material in this chapter is very important for those wanting to write highly structured code. This information is also important to those who want to understand how block structured high level languages like Pascal and Ada operate.

5.2 Lexical Nesting, Static Links, and Displays

In block structured languages like Pascal[1] it is possible to nest procedures and functions. Nesting one procedure within another limits the access to the nested procedure; you cannot access the nested procedure from outside the enclosing procedure. Likewise, variables you declare within a procedure are visible inside that procedure and to all procedures nested within that procedure[2]. This is the standard block structured language notion of scope that should be quite familiar to anyone who has written Pascal or Ada programs.

There is a good deal of complexity hidden behind the concept of scope, or lexical nesting, in a block structured language. While accessing a local variable in the current activation record is efficient, accessing global variables in a block structured language can be very inefficient. This section will describe how a high level language like Pascal deals with non-local identifiers and how to access global variables and call non-local procedures and functions.

5.2.1 Scope

Scope in most high level languages is a static, or compile-time concept[3]. Scope is the notion of when a name is visible, or accessible, within a program. This ability to hide names is useful in a program because it is often convenient to reuse certain (non-descriptive) names. The *i* variable used to control most FOR loops in high level languages is a perfect example.

The scope of a name limits its visibility within a program. That is, a program has access to a variable name only within that name's scope. Outside the scope, the program cannot access that name. Many programming languages, like Pascal and C++, allow you to reuse identifiers if the scopes of those multiple uses do not overlap. As you've seen, HLA provides scoping features for its variables. There is, however, another issue related to scope: *address binding* and *variable lifetime*. Address binding is the process of associating a memory address with a variable name. Variable lifetime is that portion of a program's execution during which a memory location is bound to a variable. Consider the following Pascal procedures:

```
procedure One(Entry:integer);
var
    i,j:integer;

    procedure Two(Parm:integer);
    var j:integer;
    begin

        for j:= 0 to 5 do writeln(i+j);
```

1. Note that C and C++ are not block structured languages. Other block structured languages include Algol, Ada, and Modula-2.
2. Subject, of course, to the limitation that you not reuse the identifier within the nested procedure.
3. There are languages that support dynamic, or run-time, scope; this text will not consider such languages.

```
            if Parm < 10 then One(Parm+1);

        end;

    begin {One}
        for i := 1 to 5 do Two(Entry);
    end;
```

Figure 5.1 shows the scope of identifiers *One, Two, Entry, i, j,* and *Parm*. The local variable *j* in procedure *Two* masks the identifier *j* in procedure *One* for statement inside procedure *Two*.

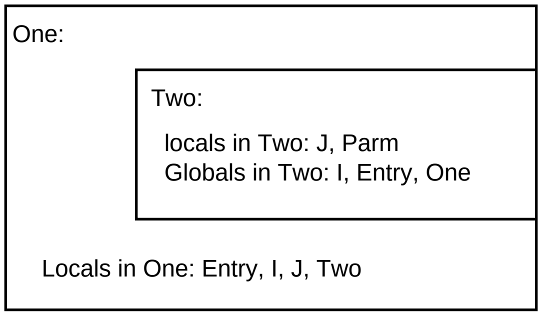
One:
Two:
locals in Two: J, Parm
Globals in Two: I, Entry, One
Locals in One: Entry, I, J, Two

Figure 5.1 Lexical Scope for Variables in Nested Pascal Procedures

5.2.2 Unit Activation, Address Binding, and Variable Lifetime

Unit activation is the process of calling a procedure or function. The combination of an activation record and some executing code is considered an *instance* of a routine. When unit activation occurs a routine binds machine addresses to its local variables. Address binding (for local variables) occurs when the routine adjusts the stack pointer to make room for the local variables. The lifetime of those variables is from that point until the routine destroys the activation record eliminating the local variable storage.

Although scope limits the visibility of a name to a certain section of code and does not allow duplicate names within the same scope, this does not mean that there is only one address bound to a name. It is quite possible to have several addresses bound to the same name at the same time. Consider a recursive procedure call. On each activation the procedure builds a new activation record. Since the previous instance still exists, there are now two activation records on the stack containing local variables for that procedure. As additional recursive activations occur, the system builds more activation records each with an address bound to the same name. To resolve the possible ambiguity (which address do you access when operating on the variable?), the system always manipulates the variable in the most recent activation record.

Note that procedures *One* and *Two* in the previous section are indirectly recursive. That is, they both call routines which, in turn, call themselves. Assuming the parameter to *One* is less than 10 on the initial call, this code will generate multiple activation records (and, therefore, multiple copies of the local variables) on the stack. For example, were you to issue the call *One(9)*, the stack would look like Figure 5.2 upon first encountering the end associated with the procedure *Two*.

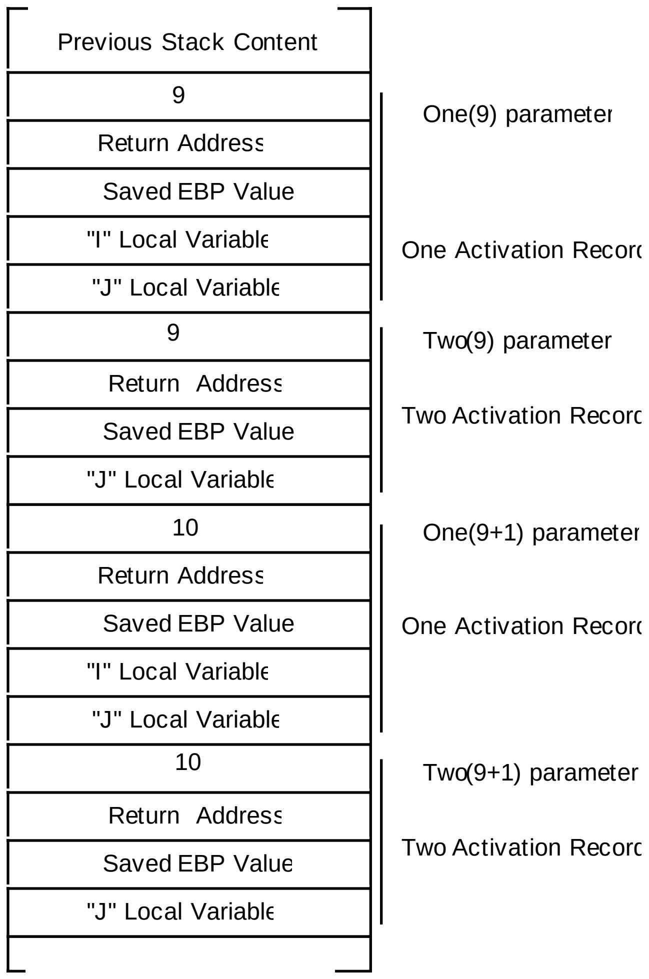

Figure 5.2 Activation Records for a Series of Recursive Calls of One and Two

As you can see, there are several copies of *I* and *J* on the stack at this point. Procedure *Two* (the currently executing routine) would access *J* in the most recent activation record that is at the bottom of Figure 5.2. The previous instance of *Two* will only access the variable *J* in its activation record when the current instance returns to *One* and then back to *Two*.

The lifetime of a variable's instance is from the point of activation record creation to the point of activation record destruction. Note that the first instance of *J* above (the one at the top of the diagram above) has the longest lifetime and that the lifetimes of all instances of *J* overlap.

5.2.3 Static Links

Pascal will allow procedure *Two* access to *I* in procedure *One*. However, when there is the possibility of recursion there may be several instances of *I* on the stack. Pascal, of course, will only let procedure *Two* access the most recent instance of *I*. In the stack diagram in Figure 5.2, this corresponds to the value of *I* in the activation record that begins with "*One(9+1)* parameter." The only problem is *how do you know where to find the activation record containing I?*

A quick, but poorly thought out answer, is to simply index backwards into the stack. After all, you can easily see in the diagram above that *I* is at offset eight from *Two's* activation record. Unfortunately, this is not always the case. Assume that procedure *Three* also calls procedure *Two* and the following statement appears within procedure *One*:

```
    If (Entry <5) then Three(Entry*2) else Two(Entry);
```

With this statement in place, it's quite possible to have two different stack frames upon entry into procedure *Two*: one with the activation record for procedure *Three* sandwiched between *One* and *Two's* activation records and one with the activation records for procedures *One* and *Two* adjacent to one another. Clearly a fixed offset from *Two's* activation record will not always point at the *I* variable on *One's* most recent activation record.

The astute reader might notice that the saved EBP value in *Two's* activation record points at the caller's activation record. You might think you could use this as a pointer to *One's* activation record. But this scheme fails for the same reason the fixed offset technique fails. EBP's old value, the dynamic link, points at the caller's activation record. Since the caller isn't necessarily the enclosing procedure the dynamic link might not point at the enclosing procedure's activation record.

What is really needed is a pointer to the enclosing procedure's activation record. Many compilers for block structured languages create such a pointer, the *static link*. Consider the following Pascal code:

```
procedure Parent;
var i,j:integer;

    procedure Child1;
    var j:integer;
    begin

        for j := 0 to 2 do writeln(i);

    end {Child1};

    procedure Child2;
    var i:integer;
    begin

        for i := 0 to 1 do Child1;

    end {Child2};

begin {Parent}

    Child2;
    Child1;

end;
```

Just after entering *Child1* for the first time, the stack would look like Figure 5.3. When *Child1* attempts to access the variable *i* from *Parent*, it will need a pointer, the static link, to *Parent's* activation record. Unfortunately, there is no way for *Child1*, upon entry, to figure out on it's own where *Parent's* activation record lies in memory. It will be necessary for the caller (*Child2* in this example) to pass the static link to *Child1*. In general, the callee can treat the static link as just another parameter; usually pushed on the stack immediately before executing the CALL instruction.

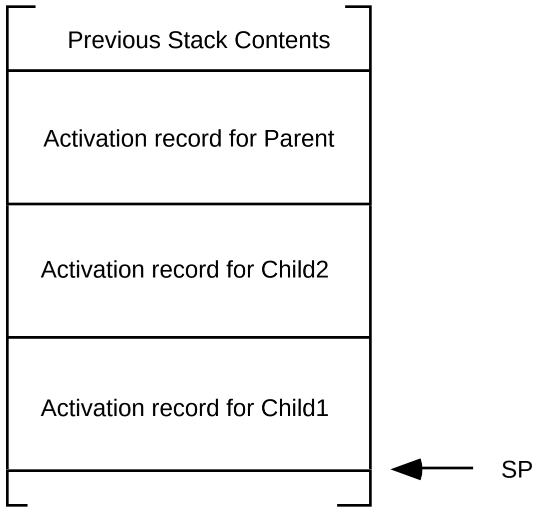
Previous Stack Contents
Activation record for Parent
Activation record for Child2
Activation record for Child1
SP

Figure 5.3 Activation Records After Several Nested Calls

To fully understand how to pass static links from call to call, you must first understand the concept of a lexical level. Lexical levels in Pascal correspond to the static nesting levels of procedures and functions. Most compiler writers specify lex level zero as the main program. That is, all symbols you declare in your main program exist at lex level zero. Procedure and function names appearing in your main program define lex level one, *no matter how many procedures or functions appear in the main program*. They all begin a new copy of lex level one. For each level of nesting, Pascal introduces a new lex level. Figure 5.4 shows this.

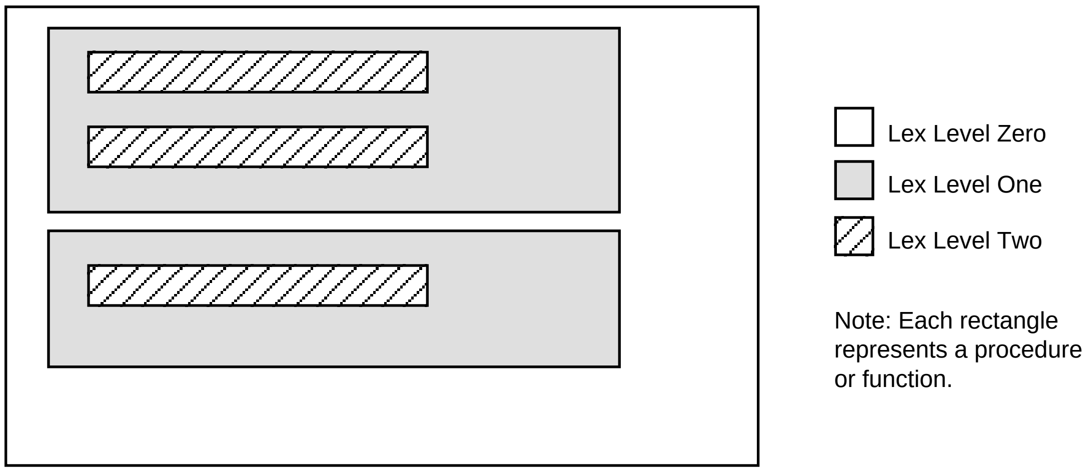

Figure 5.4 Procedure Schematic Showing Lexical Levels

During execution, a program may only access variables at a lex level less than or equal to the level of the current routine. Furthermore, only one set of values at any given lex level are accessible at any one time[4] and those values are always in the most recent activation record at that lex level.

Before worrying about how to access non-local variables using a static link, you need to figure out how to pass the static link as a parameter. When passing the static link as a parameter to a program unit (procedure or function), there are three types of calling sequences to worry about:

- A program unit calls a child procedure or function. If the current lex level is n, then a child procedure or function is at lex level n+1 and is local to the current program unit. Note that most block structured languages do not allow calling procedures or functions at lex levels greater than n+1.
- A program unit calls a peer procedure or function. A peer procedure or function is one at the same lexical level as the current caller and a single program unit encloses both program units.
- A program unit calls an ancestor procedure or function. An ancestor unit is either the parent unit, a parent of an ancestor unit, or a peer of an ancestor unit.

Calling sequences for the first two types of calls above are very simple. For the sake of this example, assume the activation record for these procedures takes the generic form in Figure 5.5.

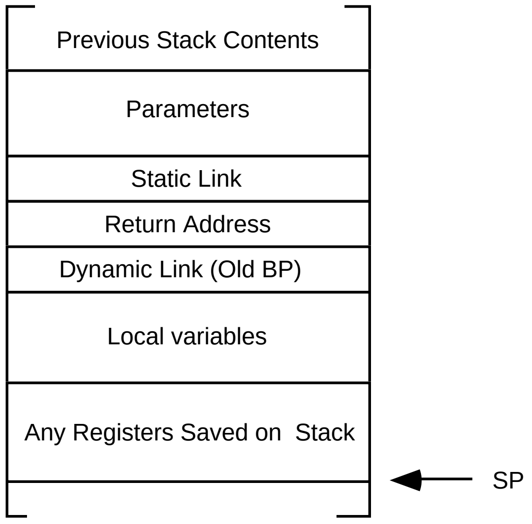

Figure 5.5 Generic Activation Record

When a parent procedure or function calls a child program unit, the static link is nothing more than the value in the EBP register immediately prior to the call. Therefore, to pass the static link to the child unit, just push EBP before executing the call instruction:

```
<Push Other Parameters onto the stack>
   push( ebp );
   call ChildUnit;
```

Of course the child unit can process the static link on the stack just like any other parameter. In this case, the static and dynamic links are exactly the same. In general, however, this is not true.

If a program unit calls a peer procedure or function, the current value in EBP is not the static link. It is a pointer to the caller's local variables and the peer procedure cannot access those variables. However, as peers, the caller and callee share the same parent program unit, so the caller can simply push a copy of its

4. There is one exception. If you have a *pointer* to a variable and the pointer remains accessible, you can access the data it points at even if the variable actually holding that data is inaccessible. Of course, in (standard) Pascal you cannot take the address of a local variable and put it into a pointer. However, certain dialects of Pascal (e.g., Turbo) and other block structured languages will allow this operation.

static link onto the stack before calling the peer procedure or function. The following code will do this assuming the current procedure’s static link is on the stack immediately above the return address:

```
<Push Other Parameters onto the Stack>
    pushd( [ebp+8] );
    call PeerUnit;
```

Calling an ancestor is a little more complex. If you are currently at lex level n and you wish to call an ancestor at lex level m (m < n), you will need to traverse the list of static links to find the desired activation record. The static links form a list of activation records. By following this chain of activation records until it ends, you can step through the most recent activation records of all the enclosing procedures and functions of a particular program unit. The stack diagram in Figure 5.6 shows the static links for a sequence of procedure calls statically nested five lex levels deep.

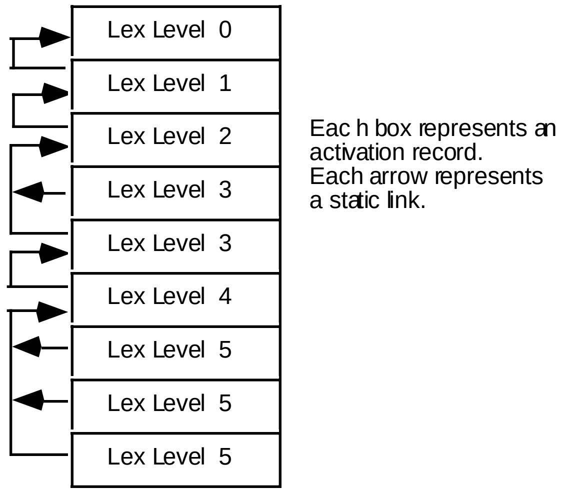

Figure 5.6 Static Links

If the program unit currently executing at lex level five wishes to call a procedure at lex level three, it must push a static link to the most recently activated program unit at lex level two. In order to find this static link you will have to *traverse* the chain of static links. If you are at lex level n and you want to call a procedure at lex level m you will have to traverse (n-m)+1 static links. The code to accomplish this is

```
// Current lex level is 5. This code locates the static link for,
// and then calls a procedure at lex level 2. Assume all calls are
// near:

    <Push necessary parameters>

    mov( [ebp+8], ebx );  // Traverse static link to LL 4.
    mov( [ebx+8], ebx );  // To Lex Level 3.
    mov( [ebx+8], ebx );  // To Lex Level 2.
    pushd( [ebx+8] );     // Ptr to most recent LL 1 activation record.
    call ProcAtLL2;
```

5.2.4 Accessing Non-Local Variables Using Static Links

In order to access a non-local variable, you must traverse the chain of static links until you get a pointer to the desired activation record. This operation is similar to locating the static link for a procedure call outlined in the previous section, except you traverse only n-m static links rather than (n-m)+1 links to obtain a pointer to the appropriate activation record. Consider the following Pascal code:

```
procedure Outer;
var i:integer;

    procedure Middle;
    var j:integer;

        procedure Inner;
        var k:integer;
        begin

            k := 3;
            writeln(i+j+k);

        end;

    begin {middle}

        j := 2;
        writeln(i+j);
        Inner;

    end; {middle}

begin {Outer}

    i := 1;
    Middle;

end; {Outer}
```

The *Inner* procedure accesses global variables at lex level n-1 and n-2 (where n is the lex level of the *Inner* procedure). The *Middle* procedure accesses a single global variable at lex level m-1 (where m is the lex level of procedure *Middle*). The following HLA code could implement these three procedures:

```
procedure Inner; @nodisplay; @noframe;
var
    k:int32;
begin Inner;

    push( ebp );
    mov( esp, ebp );
    sub( _vars_, esp );       // Make room for k.

    mov( 3, k );              // Initialize k.
    mov( [ebp+8], ebx );      // Static link to previous lex level.
    mov( [ebx-4], eax );      // Get j’s value.
    add( k, eax );            // Add in k’s value.
    mov( [ebx+8], ebx );      // Get static link to Outer’s activation record.
    add( [ebx-4], eax );      // Add in i’s value to sum.
    stdout.puti( eax );       // Display the sum.
    stdout.newln();

    mov( ebp, esp );          // Standard exit sequence.
    pop( ebp );
    ret( 4 );                // Removes the stack link from the stack.

end Inner;

procedure Middle; @nodisplay; @noframe;
var
    j:int32;
begin Middle;

    push( ebp );
    mov( esp, ebp );
    sub( _vars_, esp );     // Make room for j.

    mov( 2, j );            // Initialize j.
    mov( [ebp+8], ebx );    // Get the static link.
    mov( [ebx-4], eax );    // Get i's value.
    add( j, eax );          // Compute i+j.
    stdout.put( eax, nl );  // Display their sum.

    push( ebp );            // Static link for inner.
    call Inner;

    mov( ebp, esp );        // Standard exit sequence
    pop( ebp );
    ret( 4 );               // Removes static link from stack.
end Middle;


procedure Outer; @nodisplay; @noframe;
var
    i:int32;
begin Outer;

    push( ebp );
    mov( esp, ebp );
    sub( _vars_, esp );    // Make room for i.

    mov( 1, i );           // Give i an initial value.
    push( ebp );           // Static link for middle.
    call Middle;

    mov( ebp, esp );       // Remove local variables
    pop( ebp );
    ret( 4 );              // Removes static link.

end Outer;
```

Note that as the difference between the lex levels of the activation records increases, it becomes less and less efficient to access global variables. Accessing global variables in the previous activation record requires only one additional instruction per access, at two lex levels you need two additional instructions, etc. If you analyze a large number of Pascal programs, you will find that most of them do not nest procedures and functions and in the ones where there are nested program units, they rarely access global variables. There is one major exception, however. Although Pascal procedures and functions rarely access local variables inside other procedures and functions, they frequently access global variables declared in the main program. Since such variables appear at lex level zero, access to such variables would be as inefficient as possible when using the static links. To solve this minor problem, most 80x86 based block structured languages allocate variables at lex level zero directly in the STATIC segment and access them directly.

5.2.5 Nesting Procedures in HLA

The example in the previous treats the procedures, syntactically, as non-nested procedures and relies upon the programmer to manual handle the lexical nesting. A severe drawback to this mechanism is that it forces the programmer to manually compute the offsets of non-local variables. Although HLA does not provide automatic support for static links, HLA does allow us to nest procedures and provides some compile-time functions to help us calculate offsets into non-global activation records. Furthermore, we can treat the static link as a parameter to the procedures, so we don't have to refer to the static link using address expressions like "[ebx+8]".

Like Pascal, HLA lets you nest procedures. You may insert a procedure in the declaration section of another procedure. The Inner, Middle, and Outer procedures of the previous section could have been written in a fashion like the following:

```
procedure Outer; @nodisplay; @noframe;
var
    i:int32;

    procedure Middle; @nodisplay; @noframe;
    var
        j:int32;

        procedure Inner; @nodisplay; @noframe;
        var
            k:int32;
        begin Inner;

            << Code for the Inner procedure >>

        end Inner;

    begin Middle;

        << code for the Middle procedure >>

    end Middle;

begin Outer;

    << code for the Outer procedure >>

end Outer;
```

There are two advantages to this scheme:

1. The identifier *Inner* is local to the *Middle* procedure and is not accessible outside *Middle* (not even to *Outer*); similarly, the identifier *Middle* is local to *Outer* and is not accessible outside *Outer*. This information hiding feature lets you prevent other code from accidentally accessing these nested procedures, just as for local variables.
2. The local identifiers *i* and *j* are accessible to the nested procedures.

Before discussing how to use this feature to access non-local variables in a more reasonable fashion using static links, let's also consider the issue of the static link itself. The static link is really nothing more than a special parameter to these functions, therefore we can declare the static link as a parameter using HLA's high level procedure declaration syntax. Since the static link must always be at a fixed offset in the activation record for all procedures, the most reasonable thing to do is always make the stack link the first parameter in the list[5]; this ensures that the static link is always found at offset "+8" in the activation record. Here's the declarations above with the static links added as parameters:

```
procedure Outer( outerStaticLink:dword ); @nodisplay; @noframe;
```

```
var
    i:int32;

    procedure Middle( middleStaticLink:dword ); @nodisplay; @noframe;
    var
        j:int32;

        procedure Inner( innerStaticLink:dword ); @nodisplay; @noframe;
        var
            k:int32;
        begin Inner;

            << Code for the Inner procedure >>

        end Inner;

    begin Middle;

        << code for the Middle procedure >>

    end Middle;

begin Outer;

    << code for the Outer procedure >>

end Outer;
```

All that remains is to discuss how one references non-local (automatic) variables in this code. As you may recall from the chapter on Intermediate Procedures in Volume Four, HLA references local variables and parameters using an address expression of the form "[ebp±offset]" where offset represents the offset of the variable into the activation record (parameters typically have a positive offset, local variables have a negative offset). Indeed, we can use the HLA compile-time @offset function to access the variables without having to manually figure out the variable's offset in the activation record, e.g.,

```
    mov( [ebp+@offset( i )], eax );
```

The statement above is semantically equivalent to

```
    mov( i, eax );
```

assuming, of course, that *i* is a local variable in the current procedure.

Because HLA automatically associates the EBP register with local variables, HLA will not allow you to use a non-local variable reference in a procedure. For example, if you tried to use the statement "mov(i, eax);" in procedure *Inner* in the example above, HLA would complain that you cannot access non-local in this manner. The problem is that HLA associates EBP with automatic variables and outside the procedure in which you declare the local variable, EBP does not point at the activation record holding that variable. Hence, the instruction "mov(i, eax);" inside the *Inner* procedure would actually load *k* into EAX, not *i* (because *k* is at the same offset in *Inner's* activation record as *i* in *Outer's* activation record).

While it's nice that HLA prevents you from making the mistake of such an illegal reference, the fact remains that there needs to be some way of referring to non-local identifiers in a procedure. HLA uses the following syntax to reference a non-local, automatic, variable:

reg$_{32}$::*identifier*

5. Assuming, of course, that you're using the default Pascal calling convention. If you were using the CDECL or STDCALL calling convention, you would always make the static link the last parameter in the parameter list.

reg_{32} represents any of the 80x86's 32-bit general purpose registers and *identifier* is the non-local identifier you wish to access. HLA substitutes an address expression of the form "[reg_{32}+@offset(identifier)]" for this expression. Given this syntax, we can now rewrite the Inner, Middle, and Outer example in a high level fashion as follows:

```
procedure Outer( outerStaticLink:dword ); @nodisplay;
var
    i:int32;

    procedure Middle( middleStaticLink:dword ); @@nodisplay;
    var
        j:int32;

        procedure Inner( innerStaticLink:dword ); nodisplay;
        var
            k:int32;
        begin Inner;

            mov( 3, k );                   // Initialize k.
            mov( innerStaticLink, ebx );  // Static link to previous lex level.
            mov( ebx::j, eax );          // Get j's value.
            add( k, eax );                 // Add in k's value.

            // Get static link to Outer's activation record and
            // add in i's value:

            mov( ebx::outerStaticLink ebx );
            add( ebx::i, eax );

            // Display the results:

            stdout.puti( eax );      // Display the sum.
            stdout.newln();

        end Inner;


    begin Middle;

        mov( 2, j );                   // Initialize j.
        mov( middleStaticLink, ebx ); // Get the static link.
        mov( ebx::i, eax );          // Get i's value.
        add( j, eax );                 // Compute i+j.
        stdout.put( eax, nl );         // Display their sum.

        Inner( ebp );                  // Inner's static link is EBP.

    end Middle;


begin Outer;

    mov( 1, i );      // Give i an initial value.
    Middle( ebp );    // Static link for middle.

end Outer;
```

This example provides only a small indication of the work needed to access variables using static links. In particular, accessing *@ebx::i* in the *Inner* procedure was simplified by the fact that EBX already contained *Middle's* static link. In the typical case, it's going to take one instruction for each lex level the code traverses in order to access a given non-local automatic variable. While this might seem bad, in typical programs you rarely access non-local variables, so the situation doesn't arrive often enough to worry about.

HLA does not provide built-in support for static links. If you are going to use static links in your programs, then you must manually pass the static links as parameters to your procedures (i.e., HLA will not take care of this for you). While it is possible to modify HLA to automatically handle static links for you, HLA provides a different mechanism for accessing non-local variables - the display. To learn about displays, keep reading...

5.2.6 The Display

After reading the previous section you might get the idea that one should never use non-local variables, or limit non-local accesses to those variables declared at lex level zero. After all, it's often easy enough to put all shared variables at lex level zero. If you are designing a programming language, you can adopt the C language designer's philosophy and simply not provide block structure. Such compromises turn out to be unnecessary. There is a data structure, the *display*, that provides efficient access to any set of non-local variables.

A display is simply an array of pointers to activation records. *Display[0]* contains a pointer to the most recent activation record for lex level zero, *Display[1]* contains a pointer to the most recent activation record for lex level one, and so on. Assuming you've maintained the *Display* array in the current STATIC segment it only takes two instructions to access any non-local variable. Pictorially, the display works as shown in Figure 5.7.

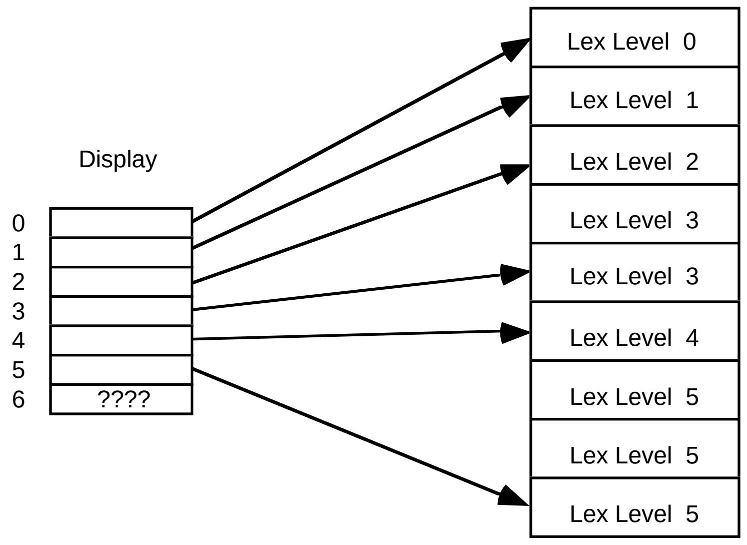

Figure 5.7 The Display

Note that the entries in the display always point at the most recent activation record for a procedure at the given lex level. If there is no active activation record for a particular lex level (e.g., lex level six above), then the entry in the display contains garbage.

The maximum lexical nesting level in your program determines how many elements there must be in the display. Most programs have only three or four nested procedures (if that many) so the display is usually quite small. Generally, you will rarely require more than 10 or so elements in the display.

Another advantage to using a display is that each individual procedure can maintain the display information itself, the caller need not get involved. When using static links the calling code has to compute and pass the appropriate static link to a procedure. Not only is this slow, but the code to do this must appear before every call. If your program uses a display, the callee, rather than the caller, maintains the display so you only need one copy of the code per procedure.

Although maintaining a single display in the STATIC segment is easy and efficient, there are a few situations where it doesn't work. In particular, when passing procedures as parameters, the single level display doesn't do the job. So for the general case, a solution other than a static array is necessary. Therefore, this chapter will not go into the details of how to maintain a static display since there are some problems with this approach.

Intel, when designing the 80286 microprocessor, studied this problem very carefully (because Pascal was popular at the time and they wanted to be able to efficiently handle Pascal constructs). They came up with a generalized solution that works for all cases. Rather than using a single display in a static segment, Intel's designers decided to have each procedure carry around its own local copy of the display. The HLA compiler automatically builds an Intel-compatible display at the beginning of each procedure, assuming you don't use the @NODISPLAY procedure option. An Intel-compatible display is part of a procedure's activation record and takes the form shown in Figure 5.8:

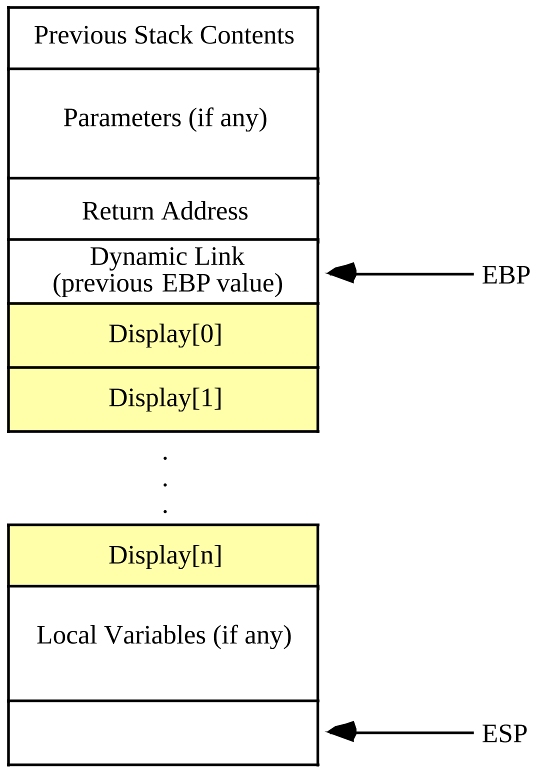

Figure 5.8 Intel-Compatible Display in an Activation Record

If we assume that the lex level of the main program is zero, then the display for a given procedure at lex level *n* will contain *n+1* double word elements. *Display[0]* is a pointer to the activation record for the main program, *Display[1]* is a pointer to the activation record of the most recently activated procedure at lex level one. Etc. *Display[n]* is a pointer to the current procedure's activation record (i.e., it contains the value found in EBP while this procedure executes). Normally, the procedure would never access element *n* of *Display* since the procedure can index off EBP directly; However, as you'll soon see, we'll need the *Display[n]* entry to build displays for procedures at higher lex levels.

One important fact to note about the Intel-compatible display array: it's elements appear backwards in memory. Remember, the stack grows downwards from high addresses to low addresses. If you study Figure 5.8 for a moment you'll discover that *Display[0]* is at the highest memory address and *Display[n]* is at the lowest memory address, exactly the opposite for standard array organization. It turns out that we'll always access the display using a constant offset, so this reversal of the array ordering is no big deal. We'll just use negative offsets from *Display[0]* (the base address of the array) rather than the usual positive offsets.

If the @NODISPLAY procedure option is not present, HLA treats the display as a predeclared local variable in the procedure and inserts the name "_display_" into the symbol table. The offset of the *_display_* variable in the activation record is the offset of the *Display[0]* entry in Figure 5.8. Therefore, you can easily access an element of this array at run-time using a statement like:

```
mov( _display_[ -lexLevel*4 ], ebx );
```

The "*4" component appears because *_display_* is an array of double words. *lexLevel* must be a constant value that specifies the lex level of the procedure whose activation record you'd like to obtain. The minus sign prefixing this expression causes HLA to index downwards in memory as appropriate for the display object.

Although it's not that difficult to figure out the lex level of a procedure manually, the HLA compile-time language provides a function that will compute the lex level of a given procedure for you – the @LEX function. This function accepts a single parameter that must be the name of an HLA procedure (that is currently in scope). The @LEX function returns an appropriate value for that function that you can use as an index into the *_display_* array. Note that @LEX returns one for the main program, two for procedures you declare in the main program, three for procedures you declare in procedures you declare in the main program, etc. If you are writing a unit, all procedures you declare in that unit exist at lex level two.

The following program is a variation of the Inner/Middle/Outer example you've seen previously in this chapter. This example uses displays and the @LEX function to access the non-local automatic variables:

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

    macro Display( proc );

        _display_[ -@lex( proc ) * 4]

    endmacro;

    procedure Outer;
    var
        i:int32;

        procedure Middle;
        var
            j:int32;

            procedure Inner;
            var
                k:int32;
            begin Inner;

                mov( 4, k );
                mov( Display( Middle ), ebx );
                mov( ebx::j, eax );            // Get j's value.
                add( k, eax );                 // Add in k's value.

                // Get static link to Outer's activation record and
                // add in i's value:

                mov( Display( Outer ), ebx );
                add( ebx::i, eax );

                // Display the results:

                stdout.puti32( eax );          // Display the sum.
                stdout.newln();

            end Inner;


        begin Middle;

            mov( 2, j );                       // Initialize j.
            mov( Display( Outer ), ebx );      // Get the static link.
            mov( ebx::i, eax );                // Get i's value.
            add( j, eax );                     // Compute i+j.
            stdout.puti32( eax );              // Display their sum.
            stdout.newln();

            Inner();

        end Middle;


    begin Outer;

        mov( 1, i );     // Give i an initial value.
        Middle();        // Static link for middle.

    end Outer;

begin DisplayDemo;

    Outer();

end DisplayDemo;
```

Program 5.1 Demonstration of Displays in an HLA Program

Assuming you do not attach the @NODISPLAY procedure option to a procedure you write in HLA, HLA will automatically emit the code (as part of the standard entry sequence) to build a display for that procedure. Up to this chapter, none of the programs in this text have used nested procedures[6], therefore there has been no need for a display. For that reason, most programs appearing in this text (since the introduction of the @NODISPLAY option) have attached @NODISPLAY to the procedure. It doesn't make a program incorrect to build a display if you never use it, but it does make the procedure a tiny bit slower and a tiny bit larger, hence the use of the @NODISPLAY option up to this point.

6. Technically, this statement is not true. Every procedure you've written has been nested inside the main program. However, none of the sample programs to date have considered the possibility of accessing the main program's automatic (VAR) variables. Hence there has been no need for a display until now).

5.2.7 The 80x86 ENTER and LEAVE Instructions

When designing the 80286, Intel's CPU designers decided to add two instructions to help maintain displays. This was done because Pascal was the popular high level language at the time and Pascal was a block structured language that could benefit from having a display. Since then, C/C++ has replaced Pascal as the most common implementation language, so these two instructions have fallen into disuse since C/C++ is not a block structured language. Still, you can take advantage of these instructions when writing assembly code with nested procedures.

Unfortunately, these two instructions, ENTER and LEAVE, are quite slow. The problem with these instructions is that C/C++ became popular shortly after Intel designed these instructions, so Intel never bothered to optimize them since few high-performance compilers actually used these instructions. On today's processors, it's actually faster to execute a sequence of instructions that do the same job than it is to actually use these instructions; hence most compilers that build displays (like HLA) emit a discrete sequence of instructions to build the display. Do keep in mind that, although these two instructions are slower than their discrete counterparts, they are generally shorter. So if you're trying to save code space rather than write the fastest possible code, using ENTER and LEAVE can help.

The LEAVE instruction is very simple to understand. It performs the same operation as the two instructions:

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

Therefore, you may use the instruction for the standard procedure exit code. On an 80386 or earlier processor, the LEAVE instruction is faster than the equivalent move and pop sequence. However, the LEAVE instruction is slower on 80486 and later processors.

The ENTER instruction takes two operands. The first is the number of bytes of local storage the current procedure requires, the second is the lex level of the current procedure. The enter instruction does the following:

```
// enter( Locals, LexLevel );

        push( ebp );          // Save dynamic link
        mov( esp, tempreg );  // Save for later.
        cmp( LexLevel, 0 );   // Done if this is lex level zero.
        je Lex0;
lp:     dec( LexLevel );
    jz Done;
        sub( 4, ebp );        // Index into display in previous activation record
        pushd( [ebp] );       //  and push the element there.
        jmp lp;

Done:
        push( tempreg );      // Add entry for current lex level.
Lex0:
        mov( tempreg, ebp );  // Pointer to current activation record.
        sub( _vars_, esp );   // Allocate storage for local variables.
```

As you can see from this code, the ENTER instruction copies the display from activation record to activation record. This can get quite expensive if you nest the procedures to any depth. Most high level languages, if they use the ENTER instruction at all, always specify a nesting level of zero to avoid copying the display throughout the stack.

The ENTER instruction puts the value for the *_display_[n]* entry at location EBP-(n*4). The ENTER instruction does not copy the value for *display[0]* into each stack frame. Intel assumes that you will keep the main program's global variables in the data segment. To save time and memory, they do not bother copying the *_display_[0]* entry. This is why HLA uses lex level one for the main program – in HLA the main program can have automatic variables and, therefore, requires a display entry.

The ENTER instruction is very slow, particularly on 80486 and later processors. If you really want to copy the display from activation record to activation record it is probably a better idea to push the items yourself. The following code snippets show how to do this:

```
// enter( n, 0 );  (n bytes of local variables, lex level zero.)

    push( ebp );       // As you can see, "enter( n, 0 );" corresponds to
    mov( esp, ebp );   //  the standard entry sequence for non-nested
    sub( n, esp );     //  procedures.

// enter( n, 1 );

    push( ebp );          // Save dynamic link (current EBP value).
    pushd( [ebp-4] );     // Push display[1] entry from previous act rec.
    lea( ebp, [esp-4] );  // Point EBP at the base of new act rec.
    sub( n, esp );        // Allocate local variables.

// enter( n, 2 );

    push( ebp );          // Save dynamic link (current EBP value).
    pushd( [ebp-4] );     // Push display[1] entry from previous act rec.
    pushd( [ebp-8] );     // Push display[2] entry from previous act rec.
    lea( ebp, [esp-8] );  // Point EBP at the base of new act rec.
    sub( n, esp );        // Allocate local variables.


// enter( n, 3 );

    push( ebp );          // Save dynamic link (current EBP value).
    pushd( [ebp-4] );     // Push display[1] entry from previous act rec.
    pushd( [ebp-8] );     // Push display[2] entry from previous act rec.
    pushd( [ebp-12] );    // Push display[3] entry from previous act rec.
    lea( ebp, [esp-12] ); // Point EBP at the base of new act rec.
    sub( n, esp );        // Allocate local variables.


// enter( n, 4 );

    push( ebp );          // Save dynamic link (current EBP value).
    pushd( [ebp-4] );     // Push display[1] entry from previous act rec.
    pushd( [ebp-8] );     // Push display[2] entry from previous act rec.
    pushd( [ebp-12] );    // Push display[3] entry from previous act rec.
    pushd( [ebp-16] );    // Push display[3] entry from previous act rec.
    lea( ebp, [esp-16] ); // Point EBP at the base of new act rec.
    sub( n, esp );        // Allocate local variables.

// etc.
```

If you are willing to believe Intel's cycle timings, you'll find that the ENTER instruction is almost never faster than a straight line sequence of instructions that accomplish the same thing. If you are interested in saving space rather than writing fast code, the ENTER instruction is generally a better alternative. The same is generally true for the LEAVE instruction as well. It is only one byte long, but it is slower than the corresponding "mov(esp, ebp);" and "pop(ebp);" instructions. The following sample program demonstrates how to access non-local variables using a display. This code does not use the @LEX function in the interest of making the lex level access clear; normally you would use the @LEX function rather than the literal constants appearing in this example.

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

```
procedure LexLevel2;

    procedure LexLevel3a;
    begin LexLevel3a;

        stdout.put( nl "LexLevel3a:" nl );
        stdout.put( "esp = ", esp, " ebp = ", ebp, nl );
        mov( _display_[0], eax );
        stdout.put( "display[0] = ", eax, nl );
        mov( _display_[-4], eax );
        stdout.put( "display[-1] = ", eax, nl );

    end LexLevel3a;

    procedure LexLevel3b; noframe;
    begin LexLevel3b;

        enter( 0, 3 );

        stdout.put( nl "LexLevel3b:" nl );
        stdout.put( "esp = ", esp, " ebp = ", ebp, nl );
        mov( _display_[0], eax );
        stdout.put( "display[0] = ", eax, nl );
        mov( _display_[-4], eax );
        stdout.put( "display[-1] = ", eax, nl );

        leave;
        ret();

    end LexLevel3b;


begin LexLevel2;

    stdout.put( "LexLevel2: esp=", esp, " ebp = ", ebp, nl nl );
    LexLevel3a();
    LexLevel3b();

end LexLevel2;

begin EnterLeaveDemo;

    stdout.put( "main: esp = ", esp, " ebp= ", ebp, nl );
    LexLevel2();

end EnterLeaveDemo;
```

Program 5.2 Demonstration of Enter and Leave in HLA

Starting with HLA v1.32, HLA provides the option of emitting ENTER or LEAVE instructions rather than the discrete sequences for a procedure's standard entry and exit sequences. The @ENTER procedure options tells HLA to emit the ENTER instruction for a procedure, the @LEAVE procedure option tells HLA to emit the LEAVE instruction in place of the standard exit sequence. See the HLA documentation for more details.

5.3 Passing Variables at Different Lex Levels as Parameters.

Accessing variables at different lex levels in a block structured program introduces several complexities to a program. The previous section introduced you to the complexity of non-local variable access. This problem gets even worse when you try to pass such variables as parameters to another program unit. The following subsections discuss strategies for each of the major parameter passing mechanisms.

For the purposes of discussion, the following sections will assume that "local" refers to variables in the current activation record, "global" refers to static variables in a static segment, and "intermediate" refers to automatic variables in some activation record other than the current activation record (this includes automatic variables in the main program). These sections will pass all parameters on the stack. You can easily modify the details to pass these parameters elsewhere, should you choose.

5.3.1 Passing Parameters by Value

Passing value parameters to a program unit is no more difficult than accessing the corresponding variables; all you need do is push the value on the stack before calling the associated procedure.

To (manually) pass a global variable by value to another procedure, you could use code like the following:

```
push( GlobalVariable );   // Assume "GlobalVariable" is a static object.
call proc;
```

To pass a local variable by value to another procedure, you could use the following code[7]:

```
push( LocalVariable );
call proc;
```

To pass an intermediate variable as a value parameter, you must first locate that intermediate variable's activation record and then push its value onto the stack. The exact mechanism you use depends on whether you are using static links or a display to keep track of the intermediate variable's activation records. If using static links, you might use code like the following to pass a variable from two lex levels up from the current procedure:

```
mov( [ebp+8], ebx );   // Assume static link is at offset 8 in Act Rec.
mov( [ebx], ebx );     // Traverse the second static link.
push( ebx::IntVar );   // Push the intermediate variable's value.
call proc;
```

Passing an intermediate variable by value when you are using a display is somewhat easier. You could use code like the following to pass an intermediate variable from lex level one:

```
mov( _display_[ -1*4 ], ebx );  // Remember each _display_ entry is 4 bytes.
push( ebx::IntVar );             // Pass the intermediate variable.
call proc;
```

It is possible to use the HLA high level procedure calling syntax when passing intermediate variables as parameters by value. The following code demonstrates this:

```
mov( _display_[ -1*4 ], ebx );
proc( ebx::IntVar );
```

This example uses a display because HLA automatically builds the display for you. If you decide to use static links, you'll have to modify this code appropriately.

7. The non-global examples all assume the variable is at offset -2 in their activation record. Change this as appropriate in your code.

5.3.2 Passing Parameters by Reference, Result, and Value-Result

The pass by reference, result, and value-result parameter mechanisms generally pass the address of parameter on the stack[8]. In an earlier chapter, you've seen how to pass global and local parameters using these mechanisms. In this section we'll take a look at passing intermediate variables by reference, value/result, and by result.

To pass an intermediate variable by reference, value/result, or by result, you must first locate the activation record containing the variable so you can compute the effective address into the stack segment. When using static links, the code to pass the parameter's address might look like the following:

```
mov( [ebp+8], ebx );       // Assume static link is at offset 8 in Act Rec.
mov( [ebx], ebx );         // Traverse the second static link.
lea( eax, ebx::IntVar );   // Get the intermediate variable's address.
push( eax );               // Pass the address on the stack.
call proc;
```

When using a display, the calling sequence might look like the following:

```
mov( _display_[ -1*4 ], ebx );  // Remember each _display_ entry is 4 bytes.
lea( eax, ebx::IntVar );        // Pass the intermediate variable.
push( eax );
call proc;
```

It is possible to use the HLA high level procedure calling syntax when passing parameters by reference, by value/result, or by result. The following code demonstrates this:

```
mov( _display_[ -1*4 ], ebx );
proc( ebx::IntVar );
```

The nice thing about the high level syntax is that it is identical whether you're passing parameters by value, reference, value/result, or by result.

As you may recall from the chapter on Low-Level Parameter Implementation, there is a second way to pass a parameter by value/result. You can push the value onto the stack and then, when the procedure returns, pop this value off the stack and store it back into the variable from whence it came. This is just a special case of the pass by value mechanism described in the previous section.

5.3.3 Passing Parameters by Name and Lazy-Evaluation in a Block Structured Language

Since you pass a thunk when passing parameters by name or by lazy-evaluation, the presence of global, intermediate, and local variables does not affect the calling sequence to the procedure. Instead, the thunk has to deal with the differing locations of these variables. Since HLA thunks already contain the pointer to the activation record for that thunk, returning a local (to the thunk) variable's address or value is especially trivial. About the only catch is what happens if you pass an intermediate variable by name or by lazy evaluation to a procedure. However, the calculation of the ultimate address (pass by name) or retrieval of the value (pass by lazy evaluation) is nearly identical to the code in the previous two sections. Hence, this code will be left as an exercise at the end of this volume.

8. As you may recall, pass by reference, value-result, and result all use the same calling sequence. The differences lie in the procedures themselves.

5.4 Passing Procedures as Parameters

Many programming languages let you pass a procedure or function name as a parameter. This lets the caller pass along various actions to perform inside a procedure. The classic example is a plot procedure that graphs some generic math function passed as a parameter to plot.

HLA lets you pass procedures and functions by declaring them as follows:

```
procedure DoCall( x:procedure );
begin DoCall;

    x();

end DoCall;
```

The statement "DoCall(xyz);" calls *DoCall* that, in turn, calls procedure *xyz*.

Whenever you pass a procedure's address in this manner, HLA only passes the address of the procedure as the parameter value. Upon entry into procedure *x* via the *DoCall* invocation, the *x* procedure first creates its own display by copying appropriate entries from *DoCall's* display. This gives *x* access to all intermediate variables that HLA allows *x* to access.

Keep in mind that thunks are special cases of functions that you call indirectly. However, there is a major difference between a thunk and a procedure – thunks carry around the pointer to the activation record they intend to use. Therefore, the thunk does not copy the calling procedure's display; instead, it uses the display of an existing procedure to access intermediate variables.

5.5 Faking Intermediate Variable Access

As you've probably noticed by now, accessing non-local (intermediate) variables is a bit less efficient than accessing local or global (static) variables. High level languages like Pascal that support intermediate variable access hide a lot of effort from the programmer that becomes painfully visible when attempting the same thing in assembly language. When attempting to write maintainable and readable code, you may want to break up a large procedure into a sequence of smaller procedures and make those smaller procedures local to a surrounding procedure that simply calls these smaller routines. Unfortunately, if the original procedure you're breaking up contains lots of local variables that code throughout the procedure shares, short of restructuring your code you will have to leave those variables in the outside procedure and access them as intermediate variables. Using the techniques of this chapter may make this task a bit unpleasant, especially if you access those variables a large number of times. This may dissuade you from attempting to break up the procedure into smaller units. Fortunately, under certain special circumstances, you can avoid the headaches of intermediate variable access in situations like this.

Consider the following short code sequence:

```
procedure MainProc;
var
    ALocalVar: dword;

    procedure proc; @nodisplay; @noframe;
    begin proc;

        mov( ebp::ALocalVar, eax );
        ret();

    end proc;

begin MainProc;

    mov( 5, ALocalVar );
    proc();

    // EAX now contains five...

end MainProc;
```

Notice that the *proc* procedure has the @NOFRAME option, so HLA does not emit the standard entry sequence to build an activation record. This means that upon entry to *proc*, EBP still points at *MainProc's* activation record. Therefore, this code can access the *ALocalVar* variable by using the syntax *ebp::ALocalVar*. No other code is necessary.

The drawback to this scheme is that proc may not contain any parameters or local variables (which would require setting EBP to point at *proc's* activation record). However, if you can live with this limitation, then this is a useful trick for accessing local variables one lex level up from the current procedure.

5.6 Putting It All Together

This chapter introduces the concept of lexical nesting commonly found in block structured languages like Pascal, Ada, and Modula-2. This chapter introduces the notion of scope, static procedure nesting, binding, variable lifetime, static links, the display, intermediate variables, and passing intermediate variables as parameters. Although few assembly programs use these features, they are occasionally useful, especially when writing code that interfaces with a high level language that supports static nesting.

Questions, Projects, and Labs Chapter Six

6.1 Questions

1) What is a *First Class Object*?
2) What is the difference between deferred and eager evaluation?
3) What is a thunk?
4) How does HLA implement thunk objects?
5) What is the purpose of the HLA THUNK statement?
6) What is the difference between a thunk and procedure variable?
7) What is the syntax for declaring a thunk as a formal parameter?
8) What is the syntax for passing a thunk constant as an actual parameter?
9) Explain how an activation record's lifetime can affect the correctness of a thunk invocation.
10) What is a trigger and how can you use a thunk to create a trigger?
11) The yield statement in an iterator isn't a true HLA statement. It's actually equivalent to something else. What is it equivalent to?
12) What is a resume frame?
13) What is the problem with breaking out of a FOREACH loop using the BREAK or BREAKIF statement?
14) What is the difference between a coroutine and a procedure?
15) What is the difference between a coroutine and a generator?
16) What is the purpose of the coret call in the coroutines class?
17) What is the limitation of a coret operation versus a standard RET instruction?
18) What is the lifetime of the automatic variables declared in a coroutine procedure?
19) Where is the easiest place to pass parameters between two coroutines?
20) Why is it difficult to pass parameters between coroutines on the stack?
21) State seven places you can pass parameters between two procedures.
22) State at least six different ways you can pass parameters.
23) Where is the most efficient place to pass parameters?
24) Where do most high level languages pass their parameters?
25) What some problems with passing parameters in global variables?
26) What is the difference between the Pascal/HLA and the CDECL parameter passing mechanisms?
27) What is the difference between the Pascal/HLA and the STDCALL parameter passing mechanisms?
28) What is the difference between the STDCALL and the CDECL parameter passing mechanisms?
29) Provide one reason why some assembly code might require the caller to remove the parameters from the stack.
30) What is the disadvantage of having the caller remove procedure parameters from the stack?
31) Explain how to pass parameters in the code stream.
32) Describe how you might pass a reference parameter in the code stream. What is the limitation on such reference parameters?

33) Explain how you might pass a “pass by value/result” or “pass by result” parameter in the code stream.

34) What is a parameter block?

35) What is the difference between pass by value/result and pass by result?

36) What is the difference between pass by name and pass by lazy evaluation?

37) What parameter passing mechanism does pass by name most closely resemble?

38) What parameter passing mechanism does pass by lazy evaluation most closely resemble?

39) When passing a parameter by name or lazy evaluation, what does HLA actually pass on the stack.

40) What is the difference in the calling sequence between pass by reference, pass by value/result, and pass by result (assuming the standard implementation)?

41) Give an example where pass by value/result produces different semantics than pass by reference.

42) What parameter passing mechanism(s) support(s) deferred execution?

For each of the following subquestions, assume that a parameter (in) is passed into one procedure and that procedure passes the parameter on to another procedure (out). Specify how to do this given the following in and out parameter passing mechanisms (if possible):

43) Parameter is passed into the first procedure by value and passed on to the second procedure by:

a. value

b. reference

c. result

d. result

e: name

f: lazy evaluation

44) Parameter is passed into the first procedure by reference and passed on to the second procedure by:

a. value

b. reference

c. result

d. result

e: name

f: lazy evaluation

45) Parameter is passed into the first procedure by value/result and passed on to the second procedure by:

a. value

b. reference

c. result

d. result

e: name

f: lazy evaluation

46) Parameter is passed into the first procedure by result and passed on to the second procedure by:

a. value

b. reference

c. result

d. result

e: name

f: lazy evaluation

47) Parameter is passed into the first procedure by name and passed on to the second procedure by:

a. value

b. reference

c. result

d. result

e: name

f: lazy evaluation

48) Parameter is passed into the first procedure by lazy evaluation and passed on to the second procedure by:

a. value

b. reference

c. result

d. result

e: name

f: lazy evaluation

49) Describe how to pass a variable number of parameters to some procedure. Describe at least two different ways to do this.

50) How can you return a function's result on the stack?

51) What's the best way to return a really large function result?

52) What is a lex level?

53) What is a static link?

54) What does the term "scope" mean?

55) What is a "display"?

56) What does the term "address binding" mean?

57) What is the "lifetime" of a variable?

58) What is an intermediate variable?

59) How do you access intermediate variables using static links? Give an example.

60) How do you access intermediate variables using a display? Give an example.

61) How do you nest procedures in HLA?

62) What does the @lex function return?

63) What is one major difference between the *_display_* array and standard arrays?

64) What does the ENTER instruction do? Provide an algorithm that describes its operation.

65) What does the LEAVE instruction do? Provide an equivalent machine code sequence.

66) Why does HLA not emit the ENTER and LEAVE instructions in those procedures that have a display?

67) Provide a short code example that demonstrates how to pass an intermediate variable by value to another procedure.

68) Provide a short code example that demonstrates how to pass an intermediate variable by reference to another procedure.

69) Provide a short code example that demonstrates how to pass an intermediate variable by value/result to another procedure.

70) Provide a short code example that demonstrates how to pass an intermediate variable by result to another procedure.

71) Provide a short code example that demonstrates how to pass an intermediate variable by name to another procedure.

72) Provide a short code example that demonstrates how to pass an intermediate variable by lazy evaluation to another procedure.

6.2 Programming Problems

1) Rewrite Program 1.1 in Chapter One (Fibonacci number generation) to use a pass by reference parameter rather than a thunk parameter.

2) Write a function ifx that has the following prototype:

```
procedure ifx( expr:boolean; lazy trueVal:dword; lazy falseVal:dword );
```

The function should test *expr's* value; if true, it should evaluate and return *trueVal*, else it should evaluate and return *falseVal*. Write a main program that tests the execution of this function.

3) Write an iterator that returns all "words" of a given length. The iterator should have the following prototype:

```
iterator wordOfLength( length:uns32 );  // returns( "eax" );
```

The iterator should allocate a string with *length* characters on the heap, initialize this string, and return a pointer to the string in the EAX register. On each call to this iterator, it should return the next string of alphabetic characters using a lexicographical ordering. E.g., for strings of length three, the iterator would return aaa, aab , aac, aad, ..., aaz, aba, abb, abc, ..., zzz. Write a main program to test this word generator. Don't forget to free the storage associated with each string in the main program when you're done with the string.

4) Modify the program in programming project (3) so that it only returns strings that have a maximum of two consenants in a row and a maximum of three vowels in a row.

5) Write a "Tic-Tac-Toe" game that uses coroutines to make each move. One coroutine should prompt the "X" player for a move, the second coroutine should prompt the "O" player for a move (note that the moves are made by players, not by the computer). The main program/coroutine should call the other two coroutines and determine if there was a win/loss/draw after each move.

6) Modify programming project (5) so that the computer makes the moves for the "O" player.

7) Write a factorial function (n!) that passes a real80 parameter on the FPU stack and returns the real80 result on the FPU stack. (note: n! = 1*2*3*...*n).

8) Write the equivalent of cs.difference that passes the two character sets to the function in the MMX registers MM0 / MM1 and MM2 / MM3. Return the character sets' difference in MM0 / MM1.

9) Write a "printstr" procedure that expects a pointer to a zero-terminated sequence of characters to follow the call to *printstr* in the code stream. This procedure should print the string to the standard output device. A typcial call will look like the following:

```
static
       staticStrVar: char; nostorage;
          byte “Hello world”, 0;
          .
          .
          .
       call printstr
       dword staticStrVar
```

Note that this function must work with any zero terminated string; don’t assume the string is an HLA string. Write a main program that makes several calls to *printstr* and tests this function.