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.

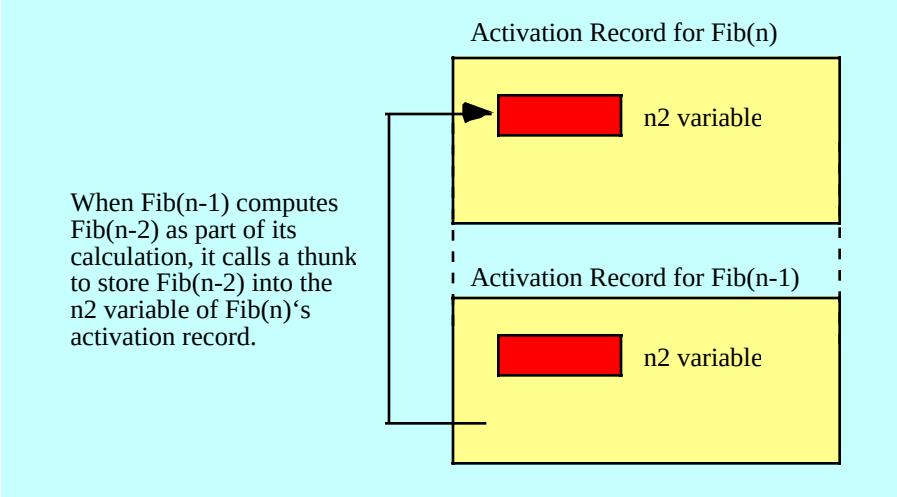

Figure 1.1 Using a Thunk to Set the Fib(n-2) Value in a Different Activation Record

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.

