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*.

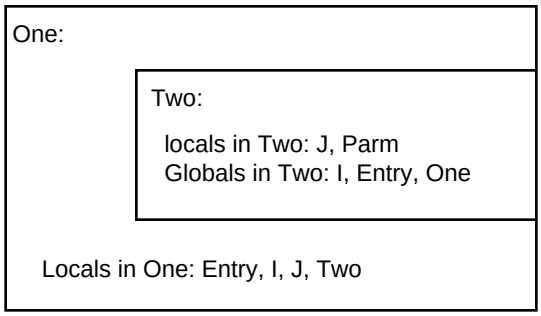

Figure 5.1 Lexical Scope for Variables in Nested Pascal Procedures

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*.

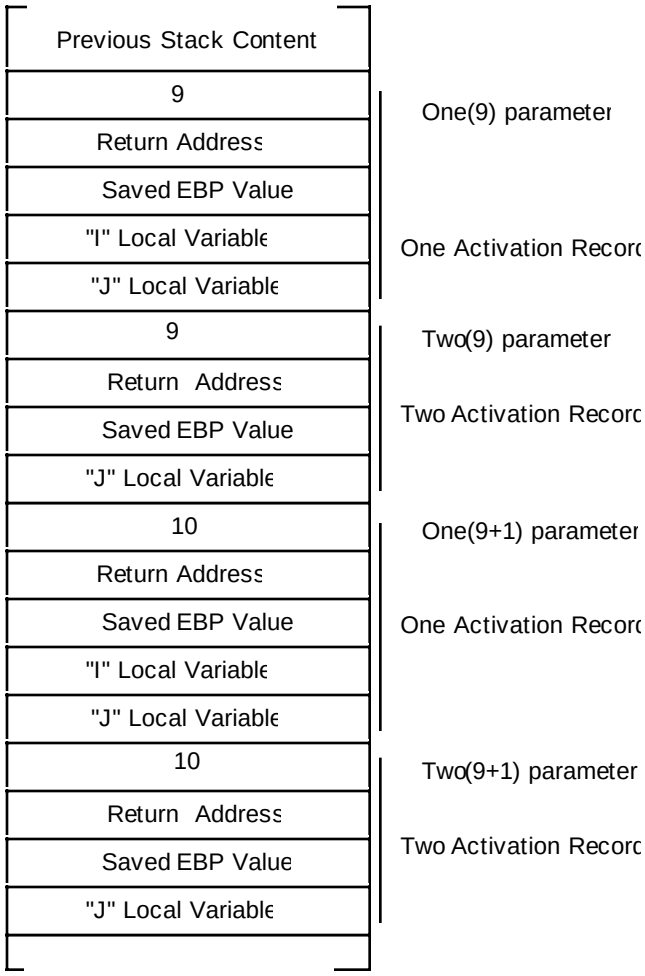

Figure 5.2 Activation Records for a Series of Recursive Calls of One and Two

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.

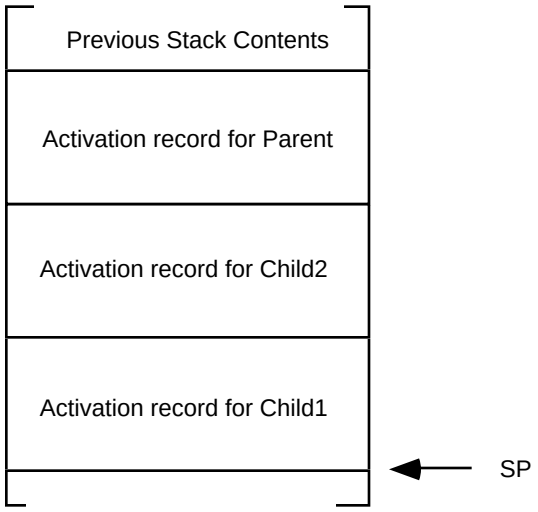

Figure 5.3 Activation Records After Several Nested Calls

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.

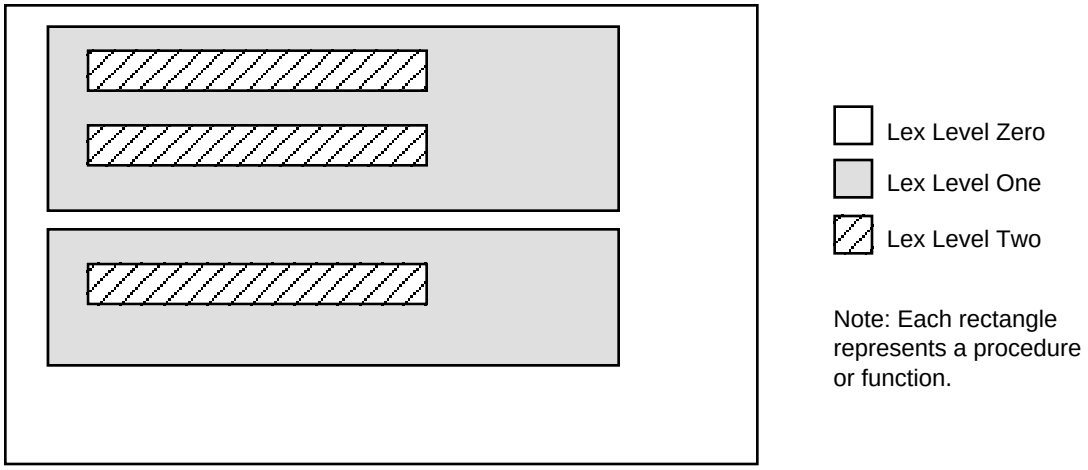

Figure 5.4 Procedure Schematic Showing Lexical Levels

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.

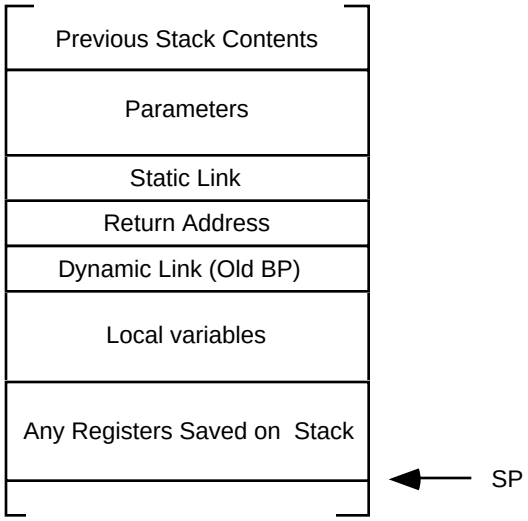

Figure 5.5 Generic Activation Record

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.

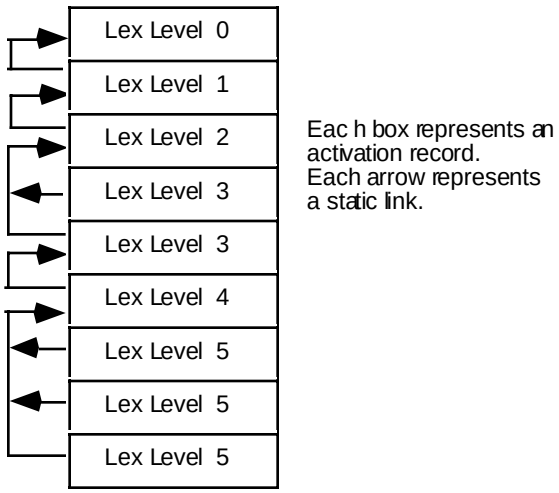

Figure 5.6 Static Links

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.

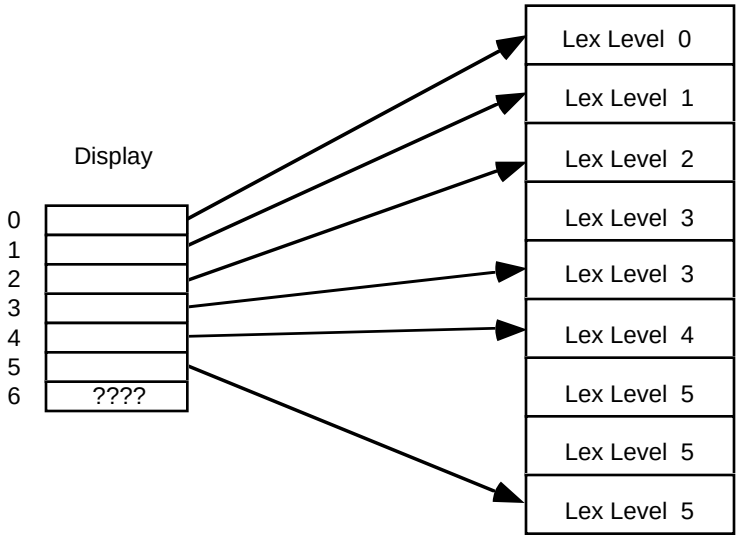

Figure 5.7 The Display

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:

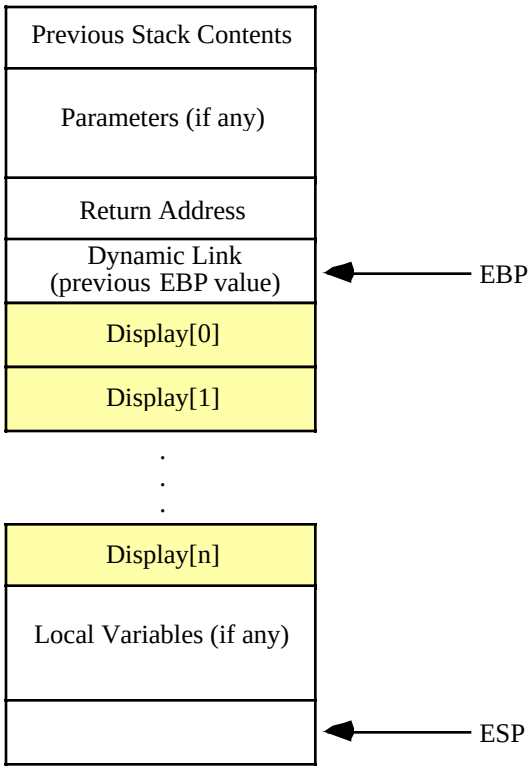

Figure 5.8 Intel-Compatible Display in an Activation Record

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.

