Procedures: Advanced Topics Chapter 12

The last chapter described how to create procedures, pass parameters, and allocate and access local variables. This chapter picks up where that one left off and describes how to access non-local variables in other procedures, pass procedures as parameters, and implement some user-defined control structures.

12.0 Chapter Overview

This chapter completes the discussion of procedures, parameters, and local variables begun in the previous chapter. This chapter describes how block structured languages like Pascal, Modula-2, Algol, and Ada access local and non-local variables. This chapter also describes how to implement a user-defined control structure, the *iterator*. Most of the material in this chapter is of interest to compiler writers and those who want to learn how compilers generate code for certain types of program constructs. Few pure assembly language programs will use the techniques this chapter describes. Therefore, none of the material in this chapter is particularly important to those who are just learning assembly language. However, if you are going to write a compiler, or you want to learn how compilers generate code so you can write efficient HLL programs, you will want to learn the material in this chapter sooner or later.

This chapter begins by discussing the notion of *scope* and how HLLs like Pascal access variables in nested procedures. The first section discusses the concept of lexical nesting and the use of static links and displays to access non-local variables. Next, this chapter discusses how to pass variables at different lex levels as parameters. The third section discusses how to pass parameters of one procedure as parameters to another procedure. The fourth major topic this chapter covers is passing procedures as parameters. This chapter concludes with a discussion of *iterators*, a user-defined control structure.

This chapter assumes a familiarity with a block structured language like Pascal or Ada. If your only HLL experience is with a non-block structured language like C, C++, BASIC, or FORTRAN, some of the concepts in this chapter may be completely new and you will have trouble understanding them. Any introductory text on Pascal or Ada will help explain any concept you don't understand that this chapter assumes is a prerequisite.

12.1 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 HLL like Pascal deals with non-local identifiers and how to access global variables and call non-local procedures and functions.

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.

12.1.1 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. Throughout this chapter you've seen equates like xyz_i, xyz_j, etc. The reason for choosing such names is that MASM doesn't support the same notion of scoped names as high level languages. Fortunately, MASM 6.x and later *does* support scoped names.

By default, MASM 6.x treats statement labels (those with a colon after them) as local to a procedure. That is, you may only reference such labels within the procedure in which they are declared. *This is true even if you nest one procedure inside another*. Fortunately, there is no good reason why anyone would want to nest procedures in a MASM program.

Having local labels within a procedure is nice. It allows you to reuse statement labels (e.g., loop labels and such) without worrying about name conflicts with other procedures. Sometimes, however, you may want to turn off the scoping of names in a procedure; a good example is when you have a case statement whose jump table appears outside the procedure. If the case statement labels are local to the procedure, they will not be visible outside the procedure and you cannot use them in the case statement jump table (see "CASE Statements" on page 525). There are two ways you can turn off the scoping of labels in MASM 6.x. The first way is to include the statement in your program:

```
option      noscoped
```

This will turn off variable scoping from that point forward in your program's source file. You can turn scoping back on with a statement of the form

```
option      scoped
```

By placing these statements around your procedure you can selectively control scoping.

Another way to control the scoping of individual names is to place a double colon ("::") after a label. This informs the assembler that this particular name should be global to the enclosing procedure.

MASM, like the C programming language, supports three levels of scope: public, global (or static), and local. Local symbols are visible only within the procedure they are defined. Global symbols are accessible throughout a source file, but are not visible in other program modules. Public symbols are visible throughout a program, across modules. MASM uses the following default scoping rules:

- By default, statement labels appearing in a procedure are local to that procedure.
- By default, all procedure names are public.
- By default, most other symbols are global.

Note that these rules apply to MASM 6.x only. Other assemblers and earlier versions of MASM follow different rules.

Overriding the default on the first rule above is easy – either use the option noscoped statement or use a double colon to make a label global. You should be aware, though, that you cannot make a local label public using the public or externdef directives. You must make the symbol global (using either technique) before you make it public.

Having all procedure names public by default usually isn't much of a problem. However, it might turn out that you want to use the same (local) procedure name in several different modules. If MASM automatically makes such names public, the linker will give you an error because there are multiple public procedures with the same name. You can turn on and off this default action using the following statements:

```
option      proc:private          ;procedures are global
```

3. There are languages that support dynamic, or run-time, scope; this text will not consider such languages.

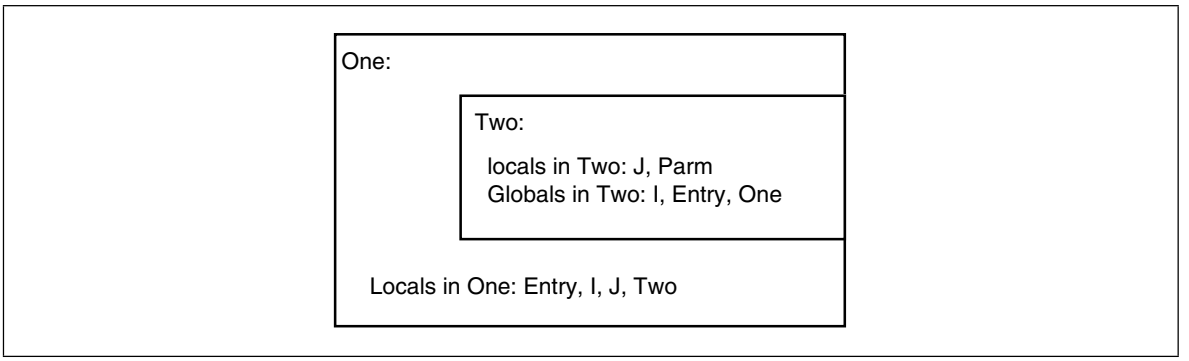

Figure 12.1 Identifier Scope

```
option      proc:export          ;procedures are public
```

Note that some debuggers only provide symbolic information if a procedure's name is public. This is why MASM 6.x defaults to public names. This problem does not exist with CodeView; so you can use whichever default is most convenient. Of course, if you elect to keep procedure names private (global only), then you will need to use the public or externdef directive to make desired procedure names public.

This discussion of local, global, and public symbols applies *mainly* to statement and procedure labels. It does *not* apply to variables you've declared in your data segment, equates, macros, typedefs, or most other symbols. Such symbols are always global regardless of where you define them. The only way to make them public is to specify their names in a public or externdef directive.

There is a way to declare parameter names and local variables, allocated on the stack, such that their names are local to a given procedure. See the proc directive in the MASM reference manual for details.

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, MASM provides some minimal scoping features for statement labels. 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);
                 if Parm < 10 then One(Parm+1);

       end;

begin {One}
       for i := 1 to 5 do Two(Entry);
end;
```

Figure 12.1 shows the scope of identifiers One, Two, Entry, i, j, and Parm.

The local variable j in Two masks the identifier j in procedure One while inside Two.

12.1.2 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 12.2 upon first encountering the end associated with the procedure 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 12.2. 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.

12.1.3 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 12.2, 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.

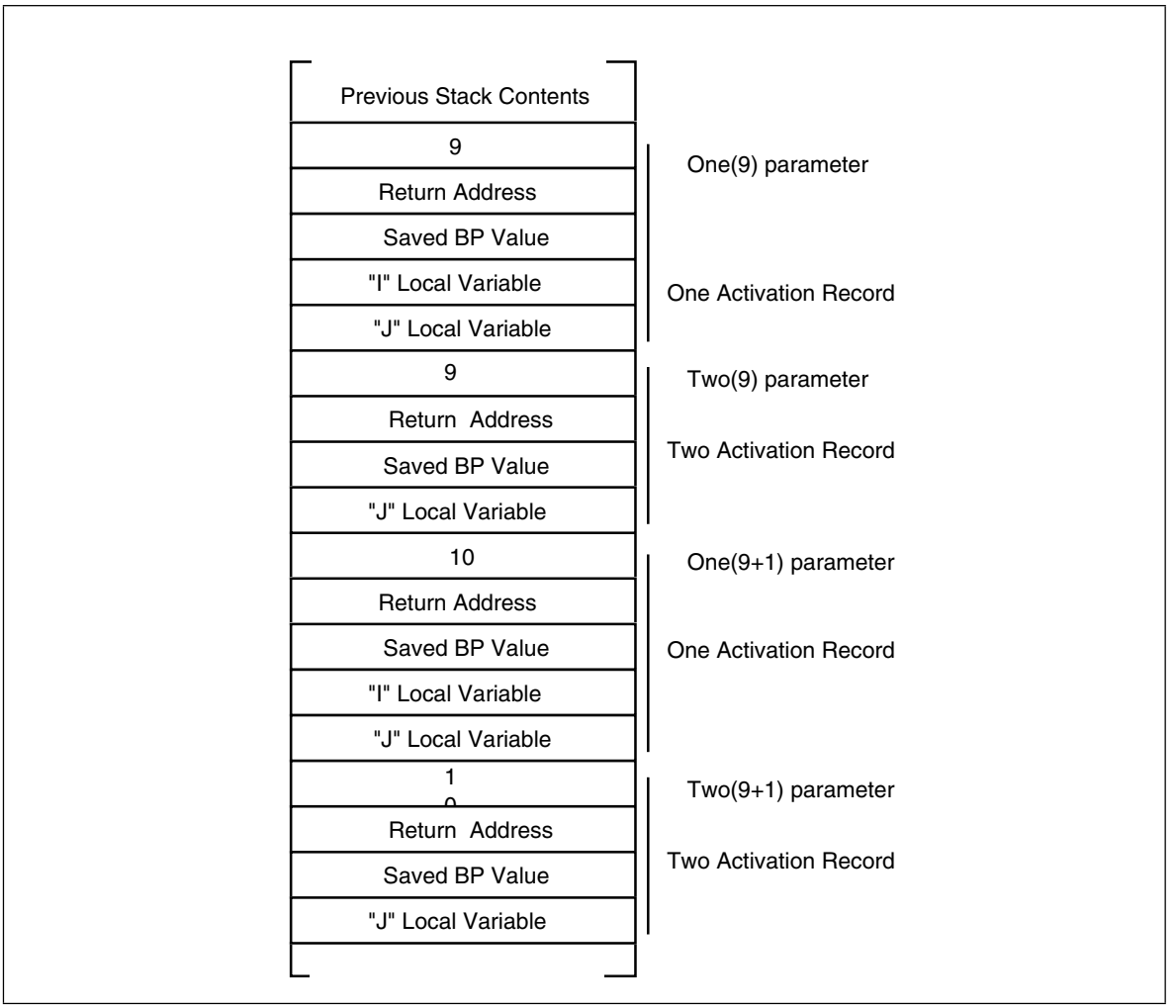

Figure 12.2 Indirect Recursion

The astute reader might notice that the saved bp 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. Bp'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};
```

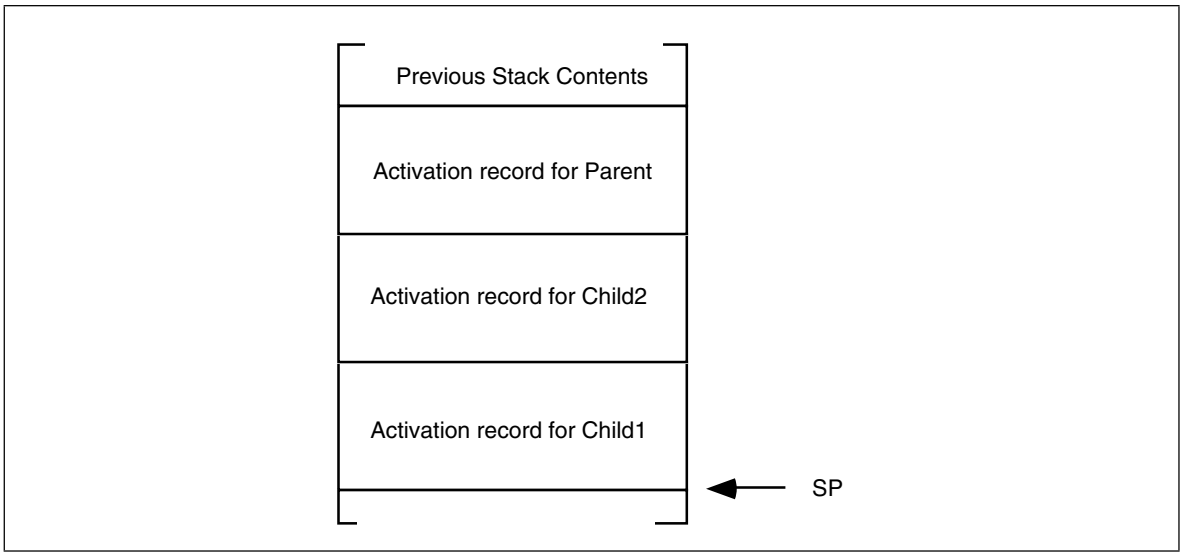

Figure 12.3 Activation Records after Several Nested Calls

```
begin {Parent}

       Child2;
       Child1;

end;
```

Just after entering Child1 for the first time, the stack would look like Figure 12.3. 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.

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 12.4 shows this.

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

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.

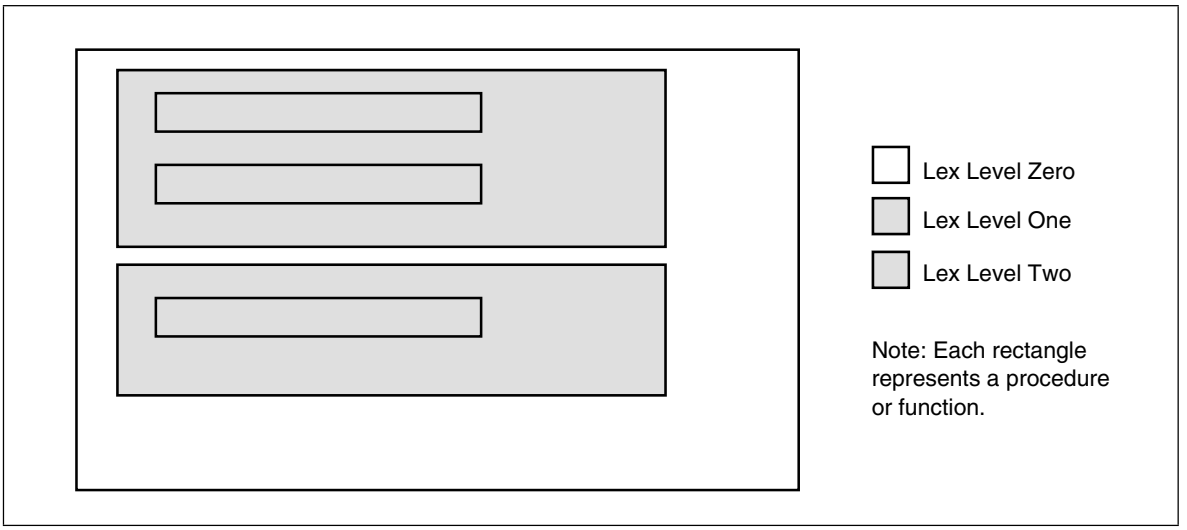

Figure 12.4 Procedure Schematic Showing Lexical Levels

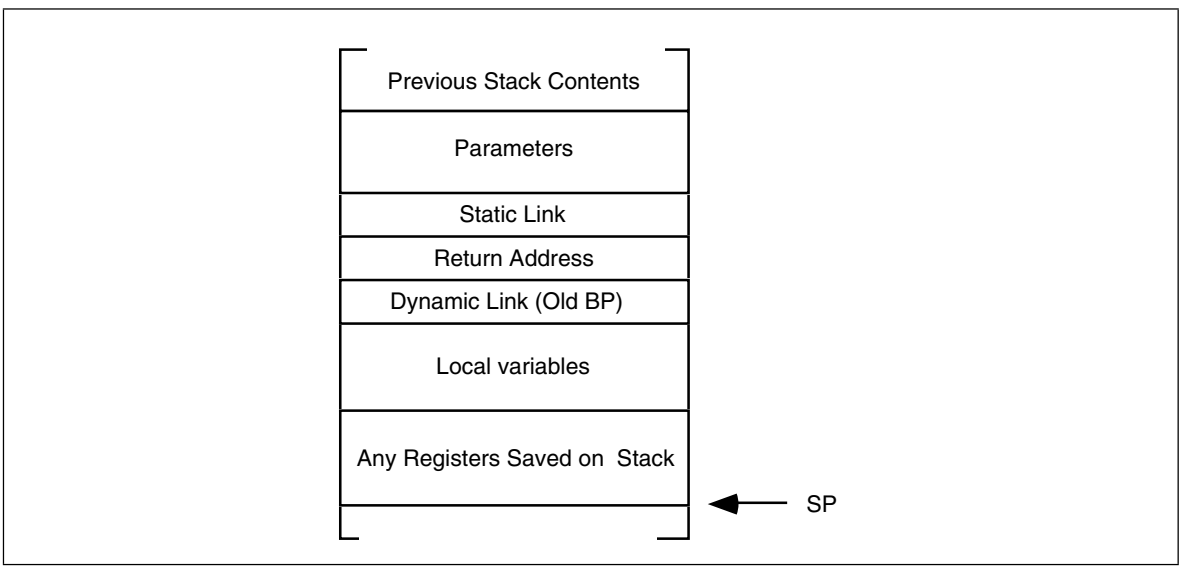

Figure 12.5 Generic Activation Record

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 12.5.

When a parent procedure or function calls a child program unit, the static link is nothing more than the value in the bp register immediately prior to the call. Therefore, to pass the static link to the child unit, just push bp before executing the call instruction:

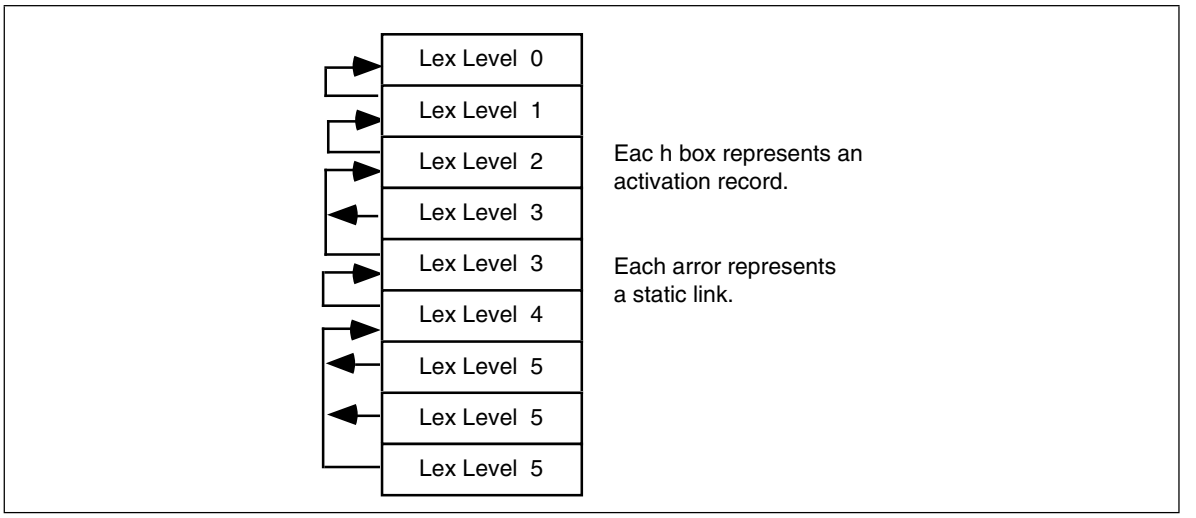

Figure 12.6 Static Links

```
<Push Other Parameters onto the stack>
          push      bp
          call      ChildUnit
```

Of course the child unit can process the static link off the stack just like any other parameter. In this case, that 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 bp 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 static link onto the stack before calling the peer procedure or function. The following code will do this, assuming all procedures and functions are near:

```
<Push Other Parameters onto the Stack>
          push      [bp+4]                  ;Push static link onto stk.
          call      PeerUnit
```

If the procedure or function is far, the static link would be two bytes farther up the stack, so you would need to use the following code:

```
<Push Other Parameters onto the Stack>
          push      [bp+6]                  ;Push static link onto stk.
          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 12.6 shows the static links for a sequence of procedure calls statically nested five lex levels deep.

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     bx, [bp+4]    ;Traverse static link to LL 4.
                mov     bx, ss:[bx+4] ;To Lex Level 3.
                mov     bx, ss:[bx+4] ;To Lex Level 2.
                push    ss:[bx+4]     ;Ptr to most recent LL1 A.R.
                call    ProcAtLL2
```

Note the ss: prefix in the instructions above. Remember, the activation records are all in the stack segment and bx indexes the data segment by default.

12.1.4 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 assembly language code could implement these three procedures:

```
Outer           proc    near
                push    bp
                mov     bp, sp
                sub     sp, 2                   ;Make room for I.

                mov     word ptr [bp-2],1       ;Set I to one.
                push    bp                      ;Static link for Middle.
                call    Middle

                mov     sp, bp                  ;Remove local variables.
                pop     bp
                ret     2                       ;Remove static link on ret.
Outer           endp

Middle          proc    near
```

```
                        push    bp                      ;Save dynamic link
                        mov     bp, sp                  ;Set up activation record.
                        sub     sp, 2                   ;Make room for J.

                        mov     word ptr [bp-2],2       ;J := 2;
                        mov     bx, [bp+4]              ;Get static link to prev LL.
                        mov     ax, ss:[bx-2]           ;Get I's value.
                        add     ax, [bp-2]              ;Add to J and then
                        puti                            ; print the sum.
                        putcr
                        push    bp                      ;Static link for Inner.
                        call    Inner

                        mov     sp, bp
                        pop     bp
                        ret     2                       ;Remove static link on RET.
Middle                  endp

Inner                   proc    near
                        push    bp                      ;Save dynamic link
                        mov     bp, sp                  ;Set up activation record.
                        sub     sp, 2                   ;Make room for K.

                        mov     word ptr [bp-2],2       ;K := 3;
                        mov     bx, [bp+4]              ;Get static link to prev LL.
                        mov     ax, ss:[bx-2]           ;Get J's value.
                        add     ax, [bp-2]              ;Add to K

                        mov     bx, ss:[bx+4]           ;Get ptr to Outer's Act Rec.
                        add     ax, ss:[bx-2]           ;Add in I's value and then
                        puti                            ; print the sum.
                        putcr

                        mov     sp, bp
                        pop     bp
                        ret     2                       ;Remove static link on RET.
Inner                   endp
```

As you can see, accessing global variables can be very inefficient[5].

Note that as the difference between 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 data segment and access them directly.

12.1.5 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.

5. Indeed, perhaps one of the main reasons the C programming language is not block structured is because the language designers wanted to avoid inefficient access to non-local variables.

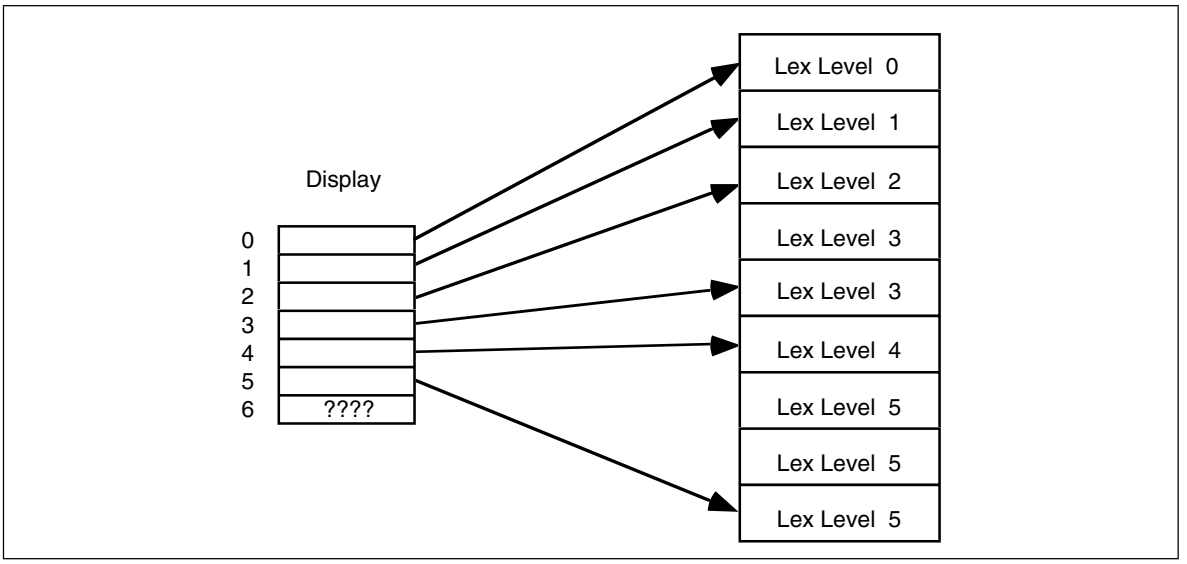

Figure 12.7 The Display

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 data segment (always a good place to keep it) it only takes two instructions to access any non-local variable. Pictorially, the display works as shown in Figure 12.7.

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. Furthermore, as the next example shows, the code to handle the display is short and fast.

Maintaining the display is very easy. Upon initial entry into a procedure you must first save the contents of the display array at the current lex level and then store the pointer to the current activation record into that same spot. Accessing a non-local variable requires only two instructions, one to load an element of the display into a register and a second to access the variable. The following code implements the Outer, Middle, and Inner procedures from the static link examples.

```
; Assume Outer is at lex level 1, Middle is at lex level 2, and
; Inner is at lex level 3. Keep in mind that each entry in the
; display is two bytes. Presumably, the variable Display is defined
; in the data segment.

Outer           proc    near
                push    bp
                mov     bp, sp
                push    Display[2]          ;Save current Display Entry
                sub     sp, 2               ;Make room for I.
```

```
                mov     word ptr [bp-4],1       ;Set I to one.
                call    Middle

                add     sp, 2                   ;Remove local variables
                pop     Display[2]              ;Restore previous value.
                pop     bp
                ret
Outer           endp

Middle          proc    near
                push    bp                      ;Save dynamic link.
                mov     bp, sp                  ;Set up our activation
record.
                push    Display[4]              ;Save old Display value.
                sub     sp, 2                   ;Make room for J.

                mov     word ptr [bp-2],2       ;J := 2;
                mov     bx, Display[2]          ;Get static link to prev LL.
                mov     ax, ss:[bx-4]           ;Get I's value.
                add     ax, [bp-2]              ;Add to J and then
                puti                            ; print the sum.
                putcr
                call    Inner

                add     sp, 2                   ;Remnove local variable.
                pop     Display[4]              ;Restore old Display value.
                pop     bp
                ret
Middle          endp

Inner           proc    near
                push    bp                      ;Save dynamic link
                mov     bp, sp                  ;Set up activation record.
                push    Display[6]              ;Save old display value
                sub     sp, 2                   ;Make room for K.

                mov     word ptr [bp-2],2       ;K := 3;
                mov     bx, Display[4]          ;Get static link to prev LL.
                mov     ax, ss:[bx-4]           ;Get J's value.
                add     ax, [bp-2]              ;Add to K

                mov     bx, Display[2]          ;Get ptr to Outer's Act Rec.
                add     ax, ss:[bx-4]           ;Add in I's value and then
                puti                            ; print the sum.
                putcr

                add     sp, 2
                pop     Display [6]
                pop     bp
                ret
Inner           endp
```

Although this code doesn't look particularly better than the former code, using a display is often much more efficient than using static links.

12.1.6 The 80286 ENTER and LEAVE Instructions

When designing the 80286, Intel's CPU designers decided to add two instructions to help maintain displays. Unfortunately, although their design works, is very general, and only requires data in the stack segment, it is very slow; much slower than using the techniques in the previous section. Although many non-optimizing compilers use these instructions, the best compilers avoid using them, if possible.

The leave instruction is very simple to understand. It performs the same operation as the two instructions:

```
                mov     sp, bp
                pop     bp
```

Therefore, you may use the instruction for the standard procedure exit code if you have an 80286 or later microprocessor. On an 80386 or earlier processor, the leave instruction is

shorter and 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    bp          ;Save dynamic link.
                mov     tempreg, sp ;Save for later.
                cmp     LexLevel, 0 ;Done if this is lex level zero.
                je      Lex0
lp:             dec     LexLevel
                jz      Done        ;Quit if at last lex level.
                sub     bp, 2       ;Index into display in prev act rec
                push    [bp]        ; and push each element there.
                jmp     lp          ;Repeat for each entry.

Done:           push    tempreg     ;Add entry for current lex level.
Lex0:           mov     bp, tempreg ;Ptr to current act rec.
                sub     sp, Locals  ;Allocate local storage
```

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 HLLs, 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 BP-(n*2). *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.

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    ;14 cycles on the 486

                push    bp          ;1 cycle on the 486
                sub     sp, n       ;1 cycle on the 486

; enter n, 1    ;17 cycles on the 486

                push    bp          ;1 cycle on the 486
                push    [bp-2]      ;4 cycles on the 486
                mov     bp, sp      ;1 cycle on the 486
                add     bp, 2       ;1 cycle on the 486
                sub     sp, n       ;1 cycle on the 486

; enter n, 2    ;20 cycles on the 486

                push    bp          ;1 cycle on the 486
                push    [bp-2]      ;4 cycles on the 486
                push    [bp-4]      ;4 cycles on the 486
                mov     bp, sp      ;1 cycle on the 486
                add     bp, 4       ;1 cycle on the 486
                sub     sp, n       ;1 cycle on the 486

; enter n, 3    ;23 cycles on the 486

                push    bp          ;1 cycle on the 486
                push    [bp-2]      ;4 cycles on the 486
                push    [bp-4]      ;4 cycles on the 486
                push    [bp-6]      ;4 cycles on the 486
                mov     bp, sp      ;1 cycle on the 486
                add     bp, 6       ;1 cycle on the 486
                sub     sp, n       ;1 cycle on the 486
```

```
; enter n, 4     ;26 cycles on the 486

                push    bp          ;1 cycle on the 486
                push    [bp-2]      ;4 cycles on the 486
                push    [bp-4]      ;4 cycles on the 486
                push    [bp-6]      ;4 cycles on the 486
                push    [bp-8]      ;4 cycles on the 486
                mov     bp, sp      ;1 cycle on the 486
                add     bp, 8       ;1 cycle on the 486
                sub     sp, n       ;1 cycle on the 486

; etc.
```

If you are willing to believe Intel's cycle timings, you can see 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 bp,sp and pop bp instructions.

Accessing non-local variables using the displays created by enter appears in the exercises.

12.2 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 variables in the data segment, and "intermediate" refers to variables in some activation record other than the current activation record. Note that the following sections will not assume that ds is equal to ss. These sections will also pass all parameters on the stack. You can easily modify the details to pass these parameters elsewhere.

12.2.1 Passing Parameters by Value in a Block Structured Language

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 pass a global variable by value to another procedure, you could use code like the following:

```
push    GlobalVar   ;Assume "GlobalVar" is in DSEG.
call    Procedure
```

To pass a local variable by value to another procedure, you could use the following code[6]:

```
push    [bp-2]      ;Local variable in current activation
call    Procedure   ; record.
```

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

6. The non-global examples all assume the variable is at offset -2 in their activation record. Change this as appropriate in your code.

code like the following to pass a variable from two lex levels up from the current procedure:

```
mov     bx, [bp+4]              ;Assume S.L. is at offset 4.
mov     bx, ss:[bx+4]           ;Traverse two static links
push    ss:[bx-2]               ;Push variables value.
call    Procedure
```

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     bx, Display[1*2]        ;Get Display[1] entry.
push    ss:[bx-2]               ;Push the variable's value.
call    Procedure
```

12.2.2 Passing Parameters by Reference, Result, and Value-Result in a Block Structured Language

The pass by reference, result, and value-result parameter mechanisms generally pass the address of parameter on the stack[7]. If global variables reside in the data segment, activation records all exist in the stack segment, and ds≠ss, then you must pass far pointers to access all possible variables[8].

To pass a far pointer you must push a segment value followed by an offset value on the stack. For global variables, the segment value is found in the ds register; for non-global values, ss contains the segment value. To compute the offset portion of the address you would normally use the lea instruction. The following code sequence passes a global variable by reference:

```
push    ds                      ;Push segment adrs first.
lea     ax, GlobalVar           ;Compute offset.
push    ax                      ;Push offset of GlobalVar
call    Procedure
```

Global variables are a special case because the assembler can compute their run-time offsets at assembly time. Therefore, *for scalar global variables only,* we can shorten the code sequence above to

```
push    ds                      ;Push segment adrs.
push    offset GlobalVar        ;Push offset portion.
call    Procedure
```

To pass a local variable by reference you code must first push ss's value onto the stack and then push the local variable's offset. *This offset is the variable's offset within the stack segment, not the offset within the activation record!* The following code passes the address of a local variable by reference:

```
push    ss                      ;Push segment address.
lea     ax, [bp-2]              ;Compute offset of local
push    ax                      ; variable and push it.
call    Procedure
```

To pass an intermediate variable by reference 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:

7. As you may recall, pass by reference, value-result, and result all use the same calling sequence. The differences lie in the procedures themselves.
8. You can use near pointers if ds=ss or if you keep global variables in the main program's activation record in the stack segment.

```
push    ss                      ;Push segment portion.
mov     bx, [bp+4]              ;Assume S.L. is at offset 4.
mov     bx, ss:[bx+4]           ;Traverse two static links
lea     ax, [bx-2]              ;Compute effective address
push    ax                      ;Push offset portion.
call    Procedure
```

When using a display, the calling sequence might look like the following:

```
push    ss                      ;Push segment portion.
mov     bx, Display[1*2]        ;Get Display[1] entry.
lea     ax, [bx-2]              ;Get the variable's offset
push    ax                      ; and push it.
call    Procedure
```

As you may recall from the previous chapter, 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.

12.2.3 Passing Parameters by Name and Lazy-Evaluation in a Block Structured Language

Since you pass the address of 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. The following examples will present thunks for pass by name, you can easily modify these thunks for lazy-evaluation parameters.

The biggest problem a thunk has is locating the activation record containing the variable whose address it returns. In the last chapter, this wasn't too much of a problem since variables existed either in the current activation record or in the global data space. In the presence of intermediate variables, this task becomes somewhat more complex. The easiest solution is to pass two pointers when passing a variable by name. The first pointer should be the address of the thunk, the second pointer should be the offset of the activation record containing the variable the thunk must access[9]. When the procedure calls the thunk, it must pass this activation record offset as a parameter to the thunk. Consider the following Panacea procedures:

```
TestThunk:procedure(name item:integer; var j:integer);
begin TestThunk;

        for j in 0..9 do item := 0;

end TestThunk;

CallThunk:procedure;
var
        A: array[0..9] : integer;
        I: integer;
endvar;
begin CallThunk;

        TestThunk(A[I], I);

end CallThunk;
```

The assembly code for the above might look like the following:

```
; TestThunk AR:
;
;       BP+10-    Address of thunk
```

9. Actually, you may need to pass several pointers to activation records. For example, if you pass the variable "A[i,j,k]" by name and A, i, j, and k are all in different activation records, you will need to pass pointers to each activation record. We will ignore this problem here.

```
;       BP+8-     Ptr to AR for Item and J parameters (must be in the same AR).
;       BP+4-     Far ptr to J.

TestThunk       proc    near
                push    bp
                mov     bp, sp
                push    ax
                push    bx
                push    es

                les     bx, [bp+4]              ;Get ptr to J.
                mov     word ptr es:[bx], 0     ;J := 0;
ForLoop:        cmp     word ptr es:[bx], 9     ;Is J > 9?
                ja      ForDone
                push    [bp+8]                  ;Push AR passed by caller.
                call    word ptr [bp+10]        ;Call the thunk.
                mov     word ptr ss:[bx], 0     ;Thunk returns adrs in BX.
                les     bx, [bp+4]              ;Get ptr to J.
                inc     word ptr es:[bx]        ;Add one to it.
                jmp     ForLoop

ForDone:        pop     es
                pop     bx
                pop     ax
                pop     bp
                ret     8
TestThunk       endp


CallThunk       proc    near
                push    bp
                mov     bp, sp
                sub     sp, 12       ;Make room for locals.

                jmp     OverThunk
Thunk           proc
                push    bp
                mov     bp, sp
                mov     bp, [bp+4]   ;Get AR address.
                mov     ax, [bp-22]  ;Get I's value.
                add     ax, ax       ;Double, since A is a word array.
                add     bx, -20      ;Offset to start of A
                add     bx, ax       ;Compute address of A[I] and
                pop     bp           ; return it in BX.
                ret     2            ;Remove parameter from stack.
Thunk           endp

OverThunk:      push    offset Thunk ;Push (near) address of thunk
                push    bp           ;Push ptr to A/I's AR for thunk
                push    ss           ;Push address of I onto stack.
                lea     ax, [bp-22]  ; Offset portion of I.
                push    ax
                call    TestThunk
                mov     sp, bp
                ret
CallThunk       endp
```

12.3 Passing Parameters as Parameters to Another Procedure

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

Pass by value parameters are essentially no different than local variables. All the techniques in the previous sections apply to pass by value parameters. The following sections

deal with the cases where the calling procedure is passing a parameter passed to it by reference, value-result, result, name, and lazy evaluation.

12.3.1 Passing Reference Parameters to Other Procedures

Passing a reference parameter though to another procedure is where the complexity begins. Consider the following (pseudo) Pascal procedure skeleton:

```
procedure HasRef(var refparm:integer);

       procedure ToProc(???? parm:integer);
       begin
          .
          .
          .
       end;

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

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

If ToProc expects a pass by value parameter (i.e., ???? is just an empty string), then HasRef needs to fetch the value of the refparm parameter and pass this value to ToProc. The following code accomplishes this[10]:

```
                les     bx, [bp+4]   ;Fetch address of refparm
                push    es:[bx]      ;Push integer pointed at by refparm
                call    ToProc
```

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

```
                push    [bp+6]       ;Push segment portion of ref parm.
                push    [bp+4]       ;Push offset portion of ref parm.
                call    ToProc
```

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

```
                jmp     SkipThunk
Thunk0          proc    near
                les     bx, [bp+4]   ;Assume BP points at HasRef's AR.
                ret
Thunk0          endp

SkipThunk:      push    offset Thunk0          ;Address of thunk.
                push    bp                     ;AR containing thunk's vars.
                call    ToProc
```

Inside ToProc, a reference to the parameter might look like the following:

```
                push    bp                     ;Save our AR ptr.
                mov     bp, [bp+4]             ;Ptr to Parm's AR.
                call    near ptr [bp+6]        ;Call the thunk.
                pop     bp                     ;Retrieve our AR ptr.
                mov     ax, es:[bx]            ;Access variable.
                  .
                  .
                  .
```

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

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

```
Thunk1          proc    near
                push    es
                push    bx
                les     bx, [bp+4]   ;Assume BP points at HasRef's AR.
                mov     ax, es:[bx]  ;Return value of ref parm in ax.
                pop     bx
                pop     es
                ret
Thunk1          endp
```

12.3.2 Passing Value-Result and Result Parameters as Parameters

Assuming you've created a local variable that holds the value of a value-result or result parameter, passing one of these parameters to another procedure is no different than passing value parameters to other code. Once a procedure makes a local copy of the value-result parameter or allocates storage for a result parameter, you can treat that variable just like a value parameter or a local variable with respect to passing it on to other procedures.

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

12.3.3 Passing Name Parameters to Other Procedures

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

When passing a name parameter as a value parameter, you first call the thunk, dereference the address the thunk returns, and then pass the value to the new procedure. The following code demonstrates such a call when the thunk returns the variable's address in es:bx (assume pass by name parameter's AR pointer is at address bp+4 and the pointer to the thunk is at address bp+6):

```
push    bp                      ;Save our AR ptr.
mov     bp, [bp+4]              ;Ptr to Parm's AR.
call    near ptr [bp+6]         ;Call the thunk.
push    word ptr es:[bx]        ;Push parameter's value.
pop     bp                      ;Retrieve our AR ptr.
call    ToProc                  ;Call the procedure.
  .
  .
  .
```

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

```
push    bp                      ;Save our AR ptr.
mov     bp, [bp+4]              ;Ptr to Parm's AR.
call    near ptr [bp+6]         ;Call the thunk.
pop     bp                      ;Retrieve our AR ptr.
push    es                      ;Push seg portion of adrs.
push    bx                      ;Push offset portion of adrs.
call    ToProc                  ;Call the procedure.
  .
  .
  .
```

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

```
push    [bp+6]              ;Pass Thunk's address.
push    [bp+4]              ;Pass adrs of Thunk's AR.
call    ToProc
```

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

12.3.4 Passing Lazy Evaluation Parameters as Parameters

Lazy evaluation parameters typically consist of three components: the address of a thunk, a location to hold the value the thunk returns, and a boolean variable that determines whether the procedure must call the thunk to get the parameter's value or if it can simply use the value previously returned by the thunk (see the exercises in the previous chapter to see how to implement lazy evaluation parameters). When passing a parameter by lazy evaluation to another procedure, the calling code must first check the boolean variable to see if the value field is valid. If not, the code must first call the thunk to get this value. If the boolean field is true, the calling code can simply use the data in the value field. In either case, once the value field has data, passing this data on to another procedure is no different than passing a local variable or a value parameter to another procedure.

12.3.5 Parameter Passing Summary

Table 48: Passing Parameters as Parameters to Another Procedure

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

Table 48: Passing Parameters as Parameters to Another Procedure

	Pass as Value	Pass as Reference	Pass as Value-Result	Pass as Result	Pass as Name	Pass as Lazy Evaluation
Name	Call the thunk, dereference the pointer, and pass the value at the address the thunk returns	Call the thunk and pass the address it returns as the parameter	Call the thunk and pass the address it returns as the parameter	Call the thunk and pass the address it returns as the parameter	Pass the address of the thunk and any other values associated with the name parameter	Write a thunk that calls the name parameter's thunk, dereferences the address it returns, and then returns the value at that address
Lazy Evaluation	If necessary, call the thunk to obtain the Lazy Eval parameter's value. Pass the local value as the value parameter	If necessary, call the thunk to obtain the Lazy Eval parameter's value. Pass the address of the local value as the parameter	If necessary, call the thunk to obtain the Lazy Eval parameter's value. Pass the address of the local value as the parameter	If necessary, call the thunk to obtain the Lazy Eval parameter's value. Pass the address of the local value as the parameter	If necessary, call the thunk to obtain the Lazy Eval parameter's value. Create a thunk that returns the address of the Lazy Eval's value field	Create a thunk that checks the boolean field of the caller's Lazy Eval parameter. It should call the corresponding thunk if this variable is false. It should set the boolean field to true and then return the data in the value field

12.4 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.

Standard Pascal lets you pass procedures and functions by declaring them as follows:

```
procedure DoCall(procedure x);
begin

       x;

end;
```

The statement DoCall(xyz); calls DoCall that, in turn, calls procedure xyz.

Passing a procedure or function as a parameter may seem like an easy task – just pass the address of the function or procedure as the following example demonstrates:

```
procedure PassMe;
begin
       Writeln('PassMe was called');
end;

procedure CallPassMe(procedure x);
begin
       x;
end;

begin {main}
       CallPassMe(PassMe);
end.
```

The 80x86 code to implement the above could look like the following:

```
PassMe          proc    near
                print
                byte    "PassMe was called",cr,lf,0
                ret
PassMe          endp

CallPassMe      proc    near
                push    bp
                mov     bp, sp
                call    word ptr [bp+4]
                pop     bp
                ret     2
CallPassMe      endp

Main            proc    near
                lea     bx, PassMe   ;Pass address of PassMe to
                push    bx           ; CallPassMe
                call    CallPassMe
                ExitPgm
Main            endp
```

For an example as simple as the one above, this technique works fine. However, it does not always work properly if PassMe needs to access non-local variables. The following Pascal code demonstrates the problem that could occur:

```
program main;

        procedure dummy;
        begin end;

        procedure Recurse1(i:integer; procedure x);

                  procedure Print;
                  begin

                           writeln(i);

                  end;

                  procedure Recurse2(j:integer; procedure y);
                  begin

                           if (j=1) then y
                           else if (j=5) then Recurse1(j-1, Print)
                           else Recurse1(j-1, y);

                  end;

        begin {Recurse1}

                  Recurse2(i, x);

        end;

begin {Main}

        Recurse1(5,dummy);

end.
```

This code produces the following call sequence:

Recurse1(5,dummy) → Recurse2(5,dummy) → Recurse1(4,Print) →
Recurse2(4,Print) → Recurse1(3,Print) → Recurse2(3,Print) →
Recurse1(2,Print) → Recurse2(2,Print) → Recurse1(1,Print) →
Recurse2(1,Print) → Print

Print will print the value of Recurse1's i variable to the standard output. However, there are several activation records present on the stack that raises the obvious question, "which copy of i does Print display?" Without giving it much thought, you might conclude that it should print the value "1" since Recurse2 calls Print when Recurse1's value for i is one. Note, though, that when Recurse2 passes the address of Print to Recurse1, i's value is four. Pascal, like most block structured languages, will use the value of i at the point Recurse2

passes the address of Print to Recurse1. Hence, the code above should print the value four, not the value one.

This creates a difficult implementation problem. After all, Print cannot simply access the display to gain access to the global variable i – the display's entry for Recurse1 points at the latest copy of Recurse1's activation record, not the entry containing the value four which is what you want.

The most common solution in systems using a display is to make a local copy of each display whenever calling a procedure or function. When passing a procedure or function as a parameter, the calling code copies the display along with the address of the procedure or function. This is why Intel's enter instruction makes a copy of the display when building the activation record.

If you are passing procedures and functions as parameters, you may want to consider using static links rather than a display. When using a static link you need only pass a single pointer (the static link) along with the routine's address. Of course, it is more work to access non-local variables, but you don't have to copy the display on every call, which is quite expensive.

The following 80x86 code provides the implementation of the above code using static links:

```
wp              textequ  <word ptr>

Dummy           proc     near
                ret
Dummy           endp

; PrintIt; (Use the name PrintIt to avoid conflict).
;
;       stack:
;
;       bp+4:     static link.

PrintIt         proc     near
                push     bp
                mov      bp, sp
                mov      bx, [bp+4]             ;Get static link
                mov      ax, ss:[bx-10]         ;Get i's value.
                puti
                pop      bp
                ret      2
PrintIt         endp

; Recurse1(i:integer; procedure x);
;
;       stack:
;
;       bp+10:    i
;       bp+8:     x's static link
;       bp+6: x's address

Recurse1        proc     near
                push     bp
                mov      bp, sp
                push     wp [bp+10]             ;Push value of i onto stack.
                push     wp [bp+8]              ;Push x's static link.
                push     wp [bp+6]              ;Push x's address.
                push     bp                     ;Push Recurse1's static link.
                call     Recurse1
                pop      bp
                ret      6
Recurse1        endp

; Recurse2(i:integer; procedure y);
;
;       stack:
;
;       bp+10: j
;       bp+8:  y's static link.
```

```
;       bp+6:  y's address.
;       bp+4:  Recurse2's static link.

Recurse2        proc    near
                push    bp
                mov     bp, sp
                cmp     wp [bp+10], 1           ;Is j=1?
                jne     TryJeq5
                push    [bp+8]                  ;y's static link.
                call    wp [bp+6]               ;Call y.
                jmp     R2Done

TryJeq5:        cmp     wp [bp+10], 5           ;Is j=5?
                jne     Call1
                mov     ax, [bp+10]
                dec     ax
                push    ax
                push    [bp+4]                  ;Push static link to R1.
                lea     ax, PrintIt             ;Push address of print.
                push    ax
                call    Recurse1
                jmp     R2Done

Call1:          mov     ax, [bp+10]
                dec     ax
                push    ax
                push    [bp+8]                  ;Pass along existing
                push    [bp+6]                  ; address and link.
                call    Recurse1

R2Done:         pop     bp
                ret     6
Recurse1        endp

main            proc
                push    bp
                mov     bp, sp
                mov     ax, 5                   ;Push first parameter.
                push    ax
                push    bp                      ;Dummy static link.
                lea     ax, Dummy               ;Push address of dummy.
                push    ax
                call    Recurse1
                pop     bp
                ExitPgm
main            endp
```

There are several ways to improve this code. Of course, this particular program doesn't really need to maintain a display or static list because only PrintIt accesses non-local variables; however, ignore that fact for the time being and pretend it does. Since you know that PrintIt only accesses variables at one particular lex level, and the program only calls PrintIt indirectly, you can pass a pointer to the appropriate activation record; this is what the above code does, although it maintains full static links as well. Compilers must always assume the worst case and often generate inefficient code. If you study your particular needs, however, you may be able to improve the efficiency of your code by avoiding much of the overhead of maintaining static lists or copying displays.

Keep in mind that thunks are special cases of functions that you call indirectly. They suffer from the same problems and drawbacks as procedure and function parameters with respect to accessing non-local variables. If such routines access non-local variables (and thunks almost always will) then you must exercise care when calling such routines. Fortunately, thunks never cause indirect recursion (which is responsible for the crazy problems in the Recurse1 / Recurse2 example) so you can use the display to access any non-local variables appearing within the thunk.

12.5 Iterators

An iterator is a cross between a control structure and a function. Although common high level languages do not often support iterators, they are present in some very high level languages[11]. Iterators provide a combination state machine/function call mechanism that lets a function pick up where it last left off on each new call. Iterators are also part of a loop control structure, with the iterator providing the value of the loop control variable on each iteration.

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

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

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

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

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

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

Variable is a variable whose type is compatible with the return type of the iterator. An iterator returns two values: a boolean *success* or *failure* value and a function result. As long as the iterator returns success, the foreach statement assigns the other return value to variable and executes statements. If iterator returns failure, the foreach loop terminates and executes the next sequential statement following the foreach loop's body. In the case of failure, the foreach statement does not affect the value of variable.

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

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

To understand how an iterator operates, consider the following short example from the Panacea programming language[12]:

```
Range:iterator(start,stop:integer):integer;
begin range;

        while (start <= stop) do

                yield start;
                start := start + 1;

        endwhile;
```

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

12. Panacea is a very high level language developed by Randall Hyde for use in compiler courses at UC Riverside.

```
end Range;
```

In the Panacea programming language, iterator calls may only appear in the foreach statement. With the exception of the *yield* statement above, anyone familiar with Pascal or C++ should be able to figure out the basic logic of this iterator.

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

```
foreach i in Range(1,10) do
        write(i);
endfor;
```

is comparable to the Pascal statement:

```
for i := 1 to 10 do write(i);
```

When a Panacea program first executes the foreach statement, it makes an *initial call* to the iterator. The iterator runs until it executes a yield or it returns. If it executes the yield statement, it returns the value of the expression following the yield as the iterator result and it returns success. If it simply returns, the iterator returns failure and no iterator result. In the current example, the initial call to the iterator returns success and the value one.

Assuming a successful return (as in the current example), the foreach statement assigns the iterator return value to the loop control variable and executes the foreach loop body. After executing the loop body, the foreach statement calls the iterator again. However, this time the foreach statement *resumes* the iterator rather than making an initial call. *An iterator resumption continues with the first statement following the last yield it executed*. In the range example, a resumption would continue execution at the start := start + 1; statement. On the first resumption, the Range iterator would add one to start, producing the value two. Two is less than ten (stop's value) so the while loop would repeat and the iterator would yield the value two. This process would repeat over and over again until the iterator yields ten. Upon resuming after yielding ten, the iterator would increment start to eleven and then return, rather than yield, since this new value is not less than or equal to ten. When the range iterator returns (fails), the foreach loop terminates.

12.5.1 Implementing Iterators Using In-Line Expansion

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

```
                push    1                       ;Assume 286 or better
                push    10                      ; and parms passed on stack.
                call    Range_Initial           ;Make initial call to iter.
                jc      Failure                 ;C=0, 1 means success, fail.
ForLoop:        puti                            ;Assume result is in AX.
                call    Range_Resume            ;Resume iterator.
                jnc     ForLoop                 ;Carry clear is success!
Failure:
```

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

```
OneToFour:iterator:integer;
begin OneToFour;

          yield 1;
          yield 2;
          yield 3;
          yield 4;

end OneToFour;
```

The initial call would execute the yield 1; statement. The first resumption would execute the yield 2; statement, the second resumption would execute yield 3;, etc. Obviously there is no single resume address the calling code can count on.

There are a couple of additional details left to consider. First, an iterator is free to call procedures and functions[13]. If such a procedure or function executes the yield statement then resumption by the foreach statement continues execution within the procedure or function that executed the yield. Second, the semantics of an iterator require all local variables and parameters to maintain their values until the iterator terminates. That is, yielding does not deallocate local variables and parameters. Likewise, any return addresses left on the stack (e.g., the call to a procedure or function that executes the yield statement) must not be lost when a piece of code yields and the corresponding foreach statement resumes the iterator. In general, this means you cannot use the standard call and return sequence to yield from or resume to an iterator because you have to preserve the contents of the stack.

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

Some high level languages support iterators in exactly this fashion. For example, Metaware's Professional Pascal Compiler for the PC supports iterators[14]. Were you to create a code sequence as follows:

```
iterator OneToFour:integer;
begin
          yield 1;
          yield 2;
          yield 3;
          yield 4;
end;
```

and call it in the main program as follows:

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

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

13. In Panacea an iterator could also call other types of program units, including other iterators, but you can ignore this for now.
14. Obviously, this is a non-standard extension to the Pascal programming language provided in Professional Pascal.

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

ForLoopCode     proc    near
                push    bp
                mov     bp, sp
                mov     ax, [bp+4]   ;Get loop control value and
                puti                 ; print it.
                putcr
                pop     bp
                ret     2            ;Pop loop control value off stk.
ForLoopCode     endp

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

                push    1            ;On 286 and later processors only.
                call    ForLoopCode
                push    2
                call    ForLoopCode
                push    3
                call    ForLoopCode
                push    4
                call    ForLoopCode
```

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

12.5.2 Implementing Iterators with Resume Frames

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

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

1) A call instruction for the initial iterator call,
2) A call instruction for the yield statement,
3) A ret instruction for the resume operation, and
4) A ret instruction to terminate the iterator.

To begin with, an iterator will require *two* return addresses rather than the single return address you would normally expect. The first return address appearing on the stack is the termination return address. The second return address is where the subroutine transfers control on a yield operation. The calling code must push these two return addresses upon initial invocation of the iterator. The stack, upon initial entry into the iterator, should look something like Figure 12.8.

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

```
foreach i in Range(1,10) do writeln(i);
```

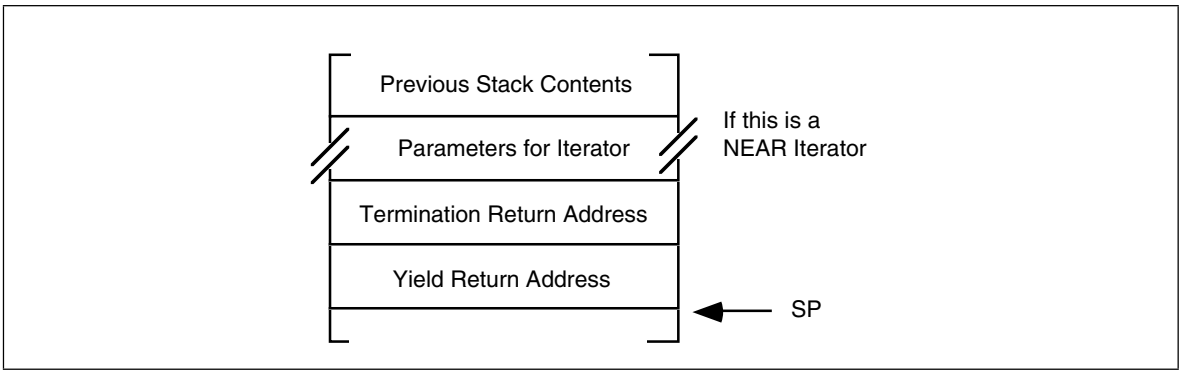

Figure 12.8 Iterator Activation Record

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

```
                push    1                       ;Push start parameter value.
                push    10                      ;Push stop parameter value.
                push    offset ForDone          ;Push termination address.
                call    Range                   ;Pushes yield return address.
```

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

```
                push    1                       ;Obviously, this requires a
                push    10                      ; 80286 or later processor.
                push    offset ForDone
                call    Range
                mov     bp, [bp]                ;Explained a little later.
                puti
                putcr
                ret
ForDone:
```

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

Now consider the Range iterator itself, here's the code to do the job:

```
Range_Start     equ     word ptr <[bp+8]>       ;Address of Start parameter.
Range_Stop      equ     word ptr <[bp+6]>       ;Address of Stop parameter.
Range_Yield     equ     word ptr <[bp+2]>       ;Yield return address.

Range           proc    near
                push    bp
                mov     bp, sp
RangeLoop:      mov     ax, Range_Start         ;Get start parameter and
                cmp     ax, Range_Stop          ; compare against stop.
                ja      RangeDone               ;Terminate if start > stop

; Okay, build the resume frame:

                push    bp                      ;Save dynamic link.
                call    Range_Yield             ;Do YIELD operation.
                pop     bp                      ;Restore dynamic link.
                inc     Range_Start             ;Bump up start value
                jmp     RangeLoop               ;Repeat until start > stop.

RangeDone:      pop     bp                      ;Restore old BP
                add     sp, 2                   ;Pop YIELD return address
                ret     4                       ;Terminate iterator.
Range           endp
```

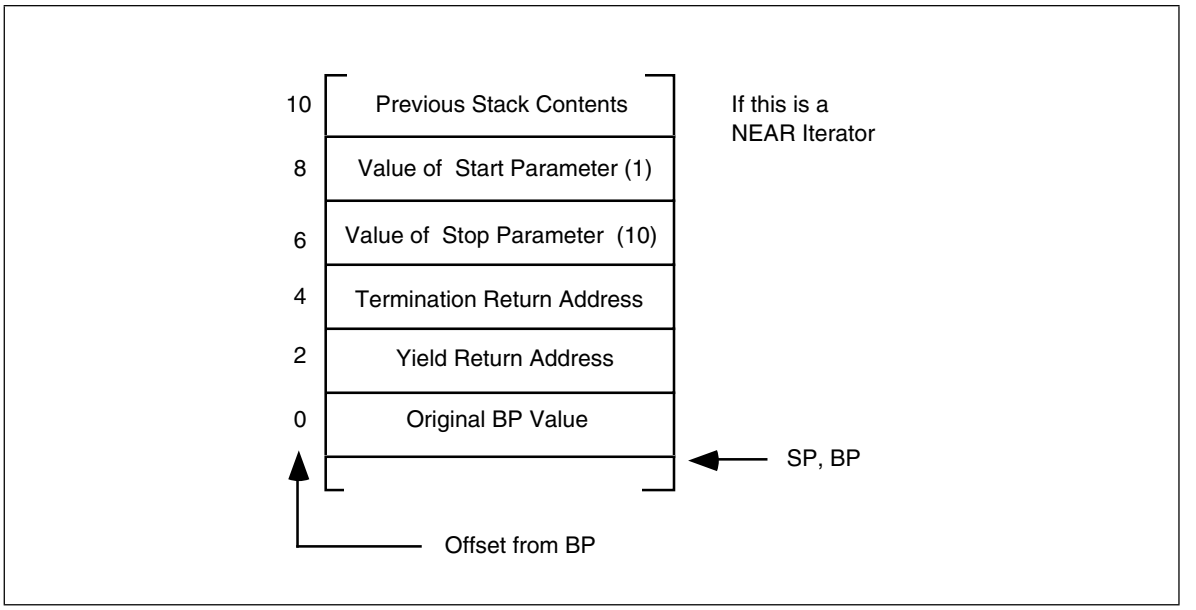

Figure 12.9 Range Activation Record

Although this routine is rather short, don't let its size deceive you; it's quite complex. The best way to describe how this iterator operates is to take it a few instructions at a time. The first two instructions are the standard entry sequence for a procedure. Upon execution of these two instructions, the stack looks like that in Figure 12.9.

The next three statements in the Range iterator, at label RangeLoop, implement the termination test of the while loop. When the Start parameter contains a value greater than the Stop parameter, control transfers to the RangeDone label at which point the code pops the value of bp off the stack, pops the yield return address off the stack (since this code will *not* return back to the body of the iterator loop) and then returns via the termination return address that is immediately above the yield return address on the stack. The return instruction also pops the two parameters off the stack.

The real work of the iterator occurs in the body of the while loop. The push, call, and pop instructions implement the yield statement. The push and call instructions build the resume frame and then return control to the body of the foreach loop. The call instruction *is not* calling a subroutine. What it is really doing here is finishing off the resume frame (by storing the yield resume address into the resume frame) and then it *returns* control back to the body of the foreach loop by jumping indirect through the yield return address pushed on the stack by the initial call to the iterator. After the execution of this call, the stack frame looks like that in Figure 12.9.

Also note that the ax register contains the return value for the iterator. As with functions, ax is a good place to return the iterator return result.

Immediately after yielding back to the foreach loop, the code must reload bp with the original value prior to the iterator invocation. This allows the calling code to correctly access parameters and local variables in its own activation record rather than the activation record of the iterator. Since bp just happens to point at the original bp value for the calling code, executing the mov bp, [bp] instruction reloads bp as appropriate. Of course, in this example reloading bp isn't necessary because the body of the foreach loop does not reference any memory locations off the bp register, but in general you will need to restore bp.

At the end of the foreach loop body the ret instruction resumes the iterator. The ret instruction pops the return address off the stack which returns control back to the iterator immediately after the call. The instruction at this point pops bp off the stack, increments the Start variable, and then repeats the while loop.

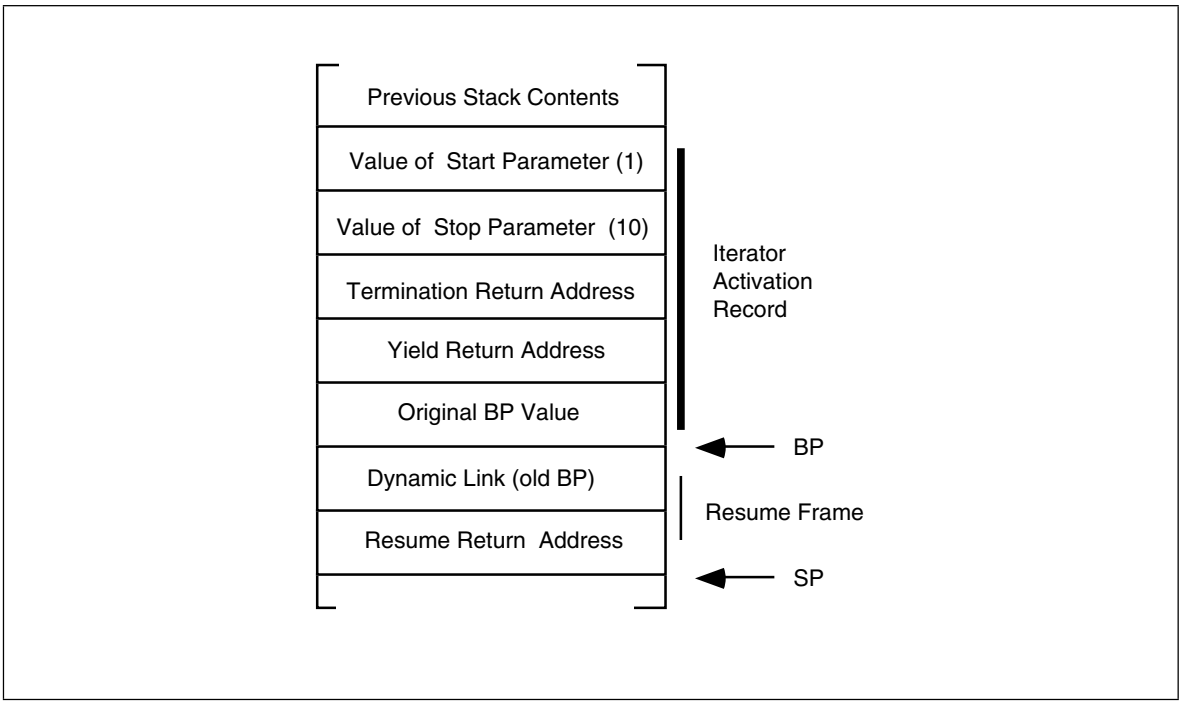

Figure 12.10 Range Resume Record

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

12.6 Sample Programs

The sample programs in this chapter provide two examples of iterators. The first example is a simple iterator that processes characters in a string and returns the vowels found in that string. The second iterator is a synthetic program (i.e., written just to demonstrate iterators) that is considerably more complex since it deals with static links. The second sample program also demonstrates another way to build the resume frame for an iterator. Take a good look at the macros that this program uses. They can simplify the user of iterators in your programs.

12.6.1 An Example of an Iterator

The following example demonstrates a simple iterator. This piece of code reads a string from the user and then locates all the vowels (a, e, i, o, u, w, y) on the line and prints their index into the string, the vowel at that position, and counts the occurrences of each vowel. This isn't a particularly good example of an iterator, however it does serve to demonstrate an implementation and use.

First, a pseudo-Pascal version of the program:

```
program DoVowels(input,output);
const
       Vowels = ['a', 'e', 'i', 'o', 'u', 'y', 'w',
                 'A', 'E', 'I', 'O', 'U', 'Y', 'W'];

var
```

```
        ThisVowel : integer;
        VowelCnt : array [char] of integer;

        iterator GetVowel(s:string) : integer;
        var
                  CurIndex : integer;
        begin

                  for CurIndex := 1 to length(s) do
                           if (s [CurIndex] in Vowels) then begin

                              { If we have a vowel, bump the cnt by 1 }
                              Vowels[s[CurIndex]] := Vowels[s[CurIndex]]+1;

                              ( Return index into string of current vowel }
                              yield CurIndex;

                           end;
        end;

begin {main}

        { First, initialize our vowel counters }

        VowelCnt ['a'] := 0;
        VowelCnt ['e'] := 0;
        VowelCnt ['i'] := 0;
        VowelCnt ['o'] := 0;
        VowelCnt ['u'] := 0;
        VowelCnt ['w'] := 0;
        VowelCnt ['y'] := 0;
        VowelCnt ['A'] := 0;
        VowelCnt ['E'] := 0;
        VowelCnt ['I'] := 0;
        VowelCnt ['O'] := 0;
        VowelCnt ['U'] := 0;
        VowelCnt ['W'] := 0;
        VowelCnt ['Y'] := 0;

        { Read and process the input string}

        Write('Enter a string: ');
        ReadLn(InputStr);
        foreach ThisVowel in GetVowel(InputStr) do
                  WriteLn('Vowel ',InputStr [ThisVowel],
                            ' at position ', ThisVowel);

        { Output the vowel counts }

        WriteLn('# of A''s:',VowelCnt['a'] + VowelCnt['A']);
        WriteLn('# of E''s:',VowelCnt['e'] + VowelCnt['E']);
        WriteLn('# of I''s:',VowelCnt['i'] + VowelCnt['I']);
        WriteLn('# of O''s:',VowelCnt['o'] + VowelCnt['O']);
        WriteLn('# of U''s:',VowelCnt['u'] + VowelCnt['U']);
        WriteLn('# of W''s:',VowelCnt['w'] + VowelCnt['W']);
        WriteLn('# of Y''s:',VowelCnt['y'] + VowelCnt['Y']);

end.
```

Here's the working assembly language version:

```
                  .286       ;For PUSH imm instr.
                  .xlist
                  include  stdlib.a
                  includelib stdlib.lib
                  .list

; Some "cute" equates:

Iterator          textequ  <proc>
endi              textequ  <endp>
wp                textequ  <word ptr>

; Necessary global variables:

dseg              segment  para public 'data'
```

```
; As per UCR StdLib instructions, InputStr must hold
; at least 128 characters.

InputStr        byte    128 dup (?)

; Note that the following statement initializes the
; VowelCnt array to zeros, saving us from having to
; do this in the main program.

VowelCnt        word    256 dup (0)

dseg            ends

cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg

; GetVowel-     This iterator searches for the next vowel in the
;               input string and returns the index to the value
;               as the iterator result. On entry, ES:DI points
;               at the string to process. On yield, AX returns
;               the zero-based index into the string of the
;               current vowel.
;
; GVYield-      Address to call when performing the yield.
; GVStrPtr-     A local variable that points at our string.

GVYield         textequ  <word ptr [bp+2]>
GVStrPtr        textequ  <dword ptr [bp-4]>

GetVowel        Iterator
                push    bp
                mov     bp, sp

; Create and initialize GVStrPtr. This is a pointer to the
; next character to process in the input string.

                push    es
                push    di

; Save original ES:DI values so we can restore them on YIELD
; and on termination.

                push    es
                push    di

; Okay, here's the main body of the iterator. Fetch each
; character until the end of the string and see if it is
; a vowel. If it is a vowel, yield the index to it. If
; it is not a vowel, move on to the next character.

GVLoop:         les     di, GVStrPtr ;Ptr to next char.
                mov     al, es:[di]  ;Get this character.
                cmp     al, 0        ;End of string?
                je      GVDone

; The following statement will convert all lower case
; characters to upper case. It will also translate other
; characters to who knows what, but we don't care since
; we only look at A, E, I, O, U, W, and Y.

                and     al, 5fh

; See if this character is a vowel. This is a disgusting
; set membership operation.

                cmp     al, 'A'
                je      IsAVowel
                cmp     al, 'E'
                je      IsAVowel
                cmp     al, 'I'
                je      IsAVowel
                cmp     al, 'O'
                je      IsAVowel
                cmp     al, 'U'
                je      IsAVowel
                cmp     al, 'W'
                je      IsAVowel
```

```
                cmp     al, 'Y'
                jne     NotAVowel

; If we've got a vowel we need to yield the index into
; the string to that vowel. To compute the index, we
; restore the original ES:DI values (which points at
; the beginning of the string) and subtract the current
; position (now in AX) from the first position. This
; produces a zero-based index into the string.
; This code must also increment the corresponding entry
; in the VowelCnt array so we can print the results
; later. Unlike the Pascal code, we've converted lower
; case to upper case so the count for upper and lower
; case characters will appear in the upper case slot.

IsAVowel:       push    bx          ;Bump the vowel
                mov     ah, 0       ; count by one.
                mov     bx, ax
                shl     bx, 1
                inc     VowelCnt[bx]
                pop     bx

                mov     ax, di
                pop     di          ;Restore original DI
                sub     ax, di      ;Compute index.
                pop     es          ;Restore original ES

                push    bp          ;Save our frame pointer
                call    GVYield     ;Yield to caller
                pop     bp          ;Restore our frame pointer
                push    es          ;Save ES:DI again
                push    di

; Whether it was a vowel or not, we've now got to move
; on to the next character in the string. Increment
; our string pointer by one and repeat the process
; over again.

NotAVowel:      inc     wp GVStrPtr
                jmp     GVLoop

; If we've reached the end of the string, terminate
; the iterator here. We need to restore the original
; ES:DI values, remove local variables, pop the YIELD
; address, and then return to the termination address.

GVDone:         pop     di          ;Restore ES:DI
                pop     es
                mov     sp, bp      ;Remove locals
                add     sp, 2       ;Pop YIELD address
                pop     bp
                ret
GetVowel        endi

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax

                print
                byte    "Enter a string: ",0
                lesi    InputStr
                gets                ;Read input line.

; The following is the foreach loop. Note that the label
; "FOREACH" is present for documentation purpose only.
; In fact, the foreach loop always begins with the first
; instruction after the call to GetVowel.
;
; One other note: this assembly language code uses
; zero-based indexes for the string. The Pascal version
; uses one-based indexes for strings. So the actual
; numbers printed will be different. If you want the
; values printed by both programs to be identical,
```

```
; uncomment the INC instruction below.

                push    offset ForDone          ;Termination address.
                call    GetVowel                ;Start iterator
FOREACH:        mov     bx, ax
                print
                byte    "Vowel ",0
                mov     al, InputStr[bx]
                putc
                print
                byte    " at position ",0
                mov     ax, bx
;               inc     ax
                puti
                putcr
                ret                             ;Iterator resume.

ForDone:        printf
                byte    "# of A's: %d\n"
                byte    "# of E's: %d\n"
                byte    "# of I's: %d\n"
                byte    "# of O's: %d\n"
                byte    "# of U's: %d\n"
                byte    "# of W's: %d\n"
                byte    "# of Y's: %d\n",0
                dword   VowelCnt + ('A'*2)
                dword   VowelCnt + ('E'*2)
                dword   VowelCnt + ('I'*2)
                dword   VowelCnt + ('O'*2)
                dword   VowelCnt + ('U'*2)
                dword   VowelCnt + ('W'*2)
                dword   VowelCnt + ('Y'*2)

Quit:           ExitPgm                         ;DOS macro to quit program.
Main            endp

cseg            ends

sseg            segment para stack 'stack'
stk             byte    1024 dup ("stack ")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       db      16 dup (?)
zzzzzzseg       ends
                end     Main
```

12.6.2 Another Iterator Example

One problem with the iterator examples appearing in this chapter up to this point is that they do not access any global or intermediate variables. Furthermore, these examples do not work if an iterator is recursive or calls other procedures that yield the value to the foreach loop. The major problem with the examples up to this point has been that the foreach loop body has been responsible for reloading the bp register with a pointer to the foreach loop's procedure's activation record. Unfortunately, the foreach loop body has to assume that bp currently points at the iterator's activation record so it can get a pointer to its own activation record from that activation record. This will not be the case if the iterator's activation record is not the one on the top of the stack.

To rectify this problem, the code doing the yield operation must set up the bp register so that it points at the activation record of the procedure containing the foreach loop before returning back to the loop. This is a somewhat complex operation. The following macro accomplishes this from inside an iterator:

```
Yield           macro
                mov     dx, [BP+2]              ;Place to yield back to.
                push    bp                      ;Save Iterator link
                mov     bp, [bp]                ;Get ptr to caller's A.R.
```

```
                call    dx                      ;Push resume address and rtn.
                pop     bp                      ;Restore ptr to our A. R.
                endm
```

Note an unfortunate side effect of this code is that it modifies the dx register. Therefore, the iterator does not preserve the dx register across a call to the iterator function.

The macro above assumes that the bp register points at the iterator's activation record. If it does not, then you must execution some additional instructions to follow the static links back to the iterator's activation record to obtain the address of the foreach loop procedure's activation record.

```
; ITERS.ASM
;
; Roughly corresponds to the example in Ghezzi & Jazayeri's
; "Programming Language Concepts" text.
;
; Randall Hyde
;
;
; This program demonstrates an implementation of:
;
;  l := 0;
;  foreach i in range(1,3) do
;      foreach j in iter2() do
;                writeln(i, ',', j, ',', l):
;
;
;  iterator range(start,stop):integer;
;  begin
;
;      while start <= stop do begin
;
;                yield start;
;                start := start+1;
;      end;
;  end;
;
;  iterator iter2:integer;
;  var k:integer;
;  begin
;
;      foreach k in iter3 do
;                yield k;
;  end;
;
;  iterator iter3:integer;
;  begin
;
;      l := l + 1;
;      yield 1;
;      l := l + 1;
;      yield 2;
;      l := l + 1;
;      yield 0;
;  end;
;
;
; This code will print:
;
;      1, 1, 1
;      1, 2, 2
;      1, 0, 3
;      2, 1, 4
;      2, 2, 5
;      2, 0, 6
;      3, 1, 7
;      3, 2, 8
;      3, 0, 9
```

```
                .xlist
                include  stdlib.a
                includelibstdlib.lib
                .list

                .286                            ;Allow extra adrs modes.

dseg            segment  para stack 'data'

; Put the stack in the data segment so we can use the small memory model
; to simplify addressing:

stk             byte     1024 dup ('stack')
EndStk          word     0

dseg            ends

cseg            segment  para public 'code'
                assume   cs:cseg, ds:dseg, ss:dseg

; Here's the structure of a resume frame.  Note that this structure isn't
; actually used in this code.  It is only provided to show you what data
; is sitting on the stack when Yield builds a resume frame.

RsmFrm          struct
ResumeAdrs      word     ?
IteratorLink    word     ?
RsmFrm          ends


; The following macro builds a resume frame and the returns to the caller
; of an iterator.  It assumes that the iterator and whoever called the
; iterator have the standard activation record defined above and that we
; are building the standard resume frame described above.
;
; This code wipes out the DX register.  Whoever calls the iterator cannot
; count on DX being preserved, likewise, the iterator cannot count on DX
; being preserved across a yield.  Presumably, the iterator returns its
; value in AX.

ActRec          struct
DynamicLink     word     ?           ;Saved BP value.
YieldAdrs       word     ?           ;Return Adrs for proc.
StaticLink      word     ?           ;Static link for proc.
ActRec          ends

AR              equ      [bp].ActRec

Yield           macro
                mov      dx, AR.YieldAdrs       ;Place to yield back to.
                push     bp                     ;Save Iterator link
                mov      bp, AR.DynamicLink     ;Get ptr to caller's A.R.
                call     dx                     ;Push resume address and rtn.
                pop      bp                     ;Restore ptr to our A. R.
                endm


; Range(start, stop) - Yields start..stop and then fails.

; The following structure defines the activation record for Range:

rngAR           struct
DynamicLink     word     ?           ;Saved BP value.
YieldAdrs       word     ?           ;Return Adrs for proc.
StaticLink      word     ?           ;Static link for proc.
FailAdrs        word     ?           ;Go here when we fail
Stop            word     ?           ;Stop parameter
Start           word     ?           ;Start parameter
```

```
rngAR           ends

rAR             equ     [bp].rngAR


Range           proc
                push    bp
                mov     bp, sp

; While start <= stop, yield start:

WhlStartLEStop: mov     ax, rAR.Start ;Also puts return value
                cmp     ax, rAR.Stop  ; in AX.
                jnle    RangeFail

                yield

                inc     rAR.Start
                jmp     WhlStartLEStop

RangeFail:      pop     bp            ;Restore Dynamic Link.
                add     sp, 4         ;Skip ret adrs and S.L.
                ret     4             ;Return through fail address.
Range           endp


; Iter2- Just calls iter3() and returns whatever value it generates.
;
; Note: Since iter2 and iter3 are at the same lex level, the static link
; passed to iter3 must be the same as the static link passed to iter2.
; This is why the "push [bp]" instruction appears below (as opposed to the
; "push bp" instruction which appears in the calls to Range and iter2).
; Keep in mind, Range and iter2 are only called from main and bp contains
; the static link at that point.  This is not true when iter2 calls iter3.

iter2           proc
                push    bp
                mov     bp, sp

                push    offset i3Fail ;Failure address.
                push    [bp]          ;Static link is link to main.
                call    iter3
                yield                 ;Return value returned by iter3
                ret                   ;Resume Iter3.

i3Fail:         pop     bp            ;Restore Dynamic Link.
                add     sp, 4         ;Skip return address & S.L.
                ret                   ;Return through fail address.
iter2           endp


; Iter3() simply yields the values 1, 2, and 0:

iter3           proc
                push    bp
                mov     bp, sp

                mov     bx, AR.StaticLink;Point BX at main's AR.
                inc     word ptr [bx-6];Increment L in main.
                mov     ax, 1
                yield

                mov     bx, AR.StaticLink
                inc     word ptr [bx-6]
                mov     ax, 2
                yield
                mov     bx, AR.StaticLink
                inc     word ptr [bx-6]
                mov     ax, 0
                yield
```

```
                pop     bp          ;Restore Dynamic Link.
                add     sp, 4       ;Skip return address & S.L.
                ret                 ;Return through fail address.
iter3           endp


; Main's local variables are allocated on the stack in order to justify
; the use of static links.

i               equ     [bp-2]
j               equ     [bp-4]
l               equ     [bp-6]

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                mov     ss, ax
                mov     sp, offset EndStk

; Allocate storage for i, j, and l on the stack:

                mov     bp, sp
                sub     sp, 6

                meminit

                mov     word ptr l, 0 ;Initialize l.

; foreach i in range(1,3) do:

                push    1             ;Parameters.
                push    3
                push    offset iFail  ;Failure address.
                push    bp          ;Static link points at our AR.
                call    Range

; Yield from range comes here.  The label is for your benefit.

RangeYield:     mov     i, ax       ;Save away loop control value.

; foreach j in iter2 do:

                push    offset jfail  ;Failure address.
                push    bp            ;Static link points at our AR.
                call    iter2

; Yield from iter2 comes here:

iter2Yield:     mov     j, ax

                mov     ax, i
                puti
                print
                byte    ", ",0
                mov     ax, j
                puti
                print
                byte    ", ",0
                mov     ax, l
                puti
                putcr

; Restart iter2:

                ret                 ;Resume iterator.

; Restart Range down here:
```

```
jFail:          ret                     ;Resume iterator.

; All Done!

iFail:          print
                byte    cr,lf,"All Done!",cr,lf,0

Quit:           ExitPgm                 ;DOS macro to quit program.
Main            endp

cseg            ends

; zzzzzzseg must be the last segment that gets loaded into memory!
; This is where the heap begins.

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       db       16 dup (?)
zzzzzzseg       ends
                end      Main
```

12.7 Laboratory Exercises

This chapter's laboratory exercises consist of three components. In the first exercise you will experiment with a fairly complex set of iterators. In the second exercise you will learn how the 80286's enter and leave instructions operate. In the third exercise, you will run some experiments on parameter passing mechanisms.

12.7.1 Iterator Exercise

In this laboratory exercise you will be working with a program (Ex12_1.asm on the companion CD-ROM) that uses four iterators. The first three iterators perform some fairly simple computations, the fourth iterator returns (successively) pointers to the first three iterators' code that the main program can use to call these iterators.

For your lab report: study the following code and explain how it works. Run it and explain the output. Assemble the program with the "/Zi" option, then from within CodeView, set a breakpoint on the first instruction of the four iterators. Run the program up to these break points and dump the memory starting at the current stack pointer value (ss:sp). Describe the meaning of the data on the stack at each breakpoint. Also, set a breakpoint on the "call ax" instruction. Trace into the routine ax points at upon each breakpoint and describe which routine this instruction calls. How many times does this instruction execute?

```
; EX12_1.asm
;
; Program to support the laboratory exercise in Chapter 12.
;
; This program combines iterators, passing parameters as parameters,
; and procedural parameters all into the same program.
;
;
; This program implements the following iterators (examples written in panacea):
;
; program EX12_1;
;
; fib:iterator(n:integer):integer;
; var
;       CurIndex:integer;
;       Fn1:        integer;
;       Fn2:        integer;
; endvar;
; begin fib;
;
```

```
;       yield 1;  (* Always have at least n=0 *)
;       if (n <> 0) then
;
;                 yield 1;  (* Have at least n=1 at this point *)
;
;                 Fn1 := 1;
;                 Fn2 := 1;
;                 foreach CurIndex in 2..n do
;
;                           yield Fn1+Fn2;
;                           Fn2 = Fn1;
;                           Fn1 = CurIndex;
;
;                 endfor;
;       endif;
;
; end fib;
;
;
;
; UpDown:iterator(n:integer):integer;
; var
;       CurIndex:integer;
; endvar;
; begin UpDown;
;
;       foreach CurIndex in 0..n do
;
;                 yield CurIndex;
;
;       endfor;
;       foreach CurIndex in n-1..0 do
;
;                 yield CurIndex;
;
;       endfor;
;
; end UpDown;
;
;
;
; SumToN:iterator(n:integer):integer;
; var
;       CurIndex:integer;
;       Sum:       integer;
; endvar;
; begin SumToN;
;
;       Sum := 0;
;       foreach CurIndex in 0..n do
;
;                 Sum := Sum + CurIndex;
;                 yield Sum;
;
;       endfor;
;
; end SumToN;
;
;
; MultiIter returns a pointer to an iterator that accepts a single integer
parameter.
;
; MultiIter: iterator: [iterator(n:integer)];
; begin MultiIter;
;
;       yield @Fib;(* Return pointers to the three iterators above *)
;       yield @UpDown;(* as the result of this iterator.*)
;       yield @SumToN;
;
; end MultiIter;
```

```
;
;
; var
;       i:integer;
;       n:integer;
;       iter:[iterator(n:integer)];
; endvar;
;
; begin EX12_1;
;
;       (* The following for loop repeats six times, passing its loop index as*)
;       (* the parameter to the Fib, UpDown, and SumToN parameters.*)
;
;       foreach n in 0..5 do
;
;
;       (* The following (funny looking) iterator sequences through *)
;       (* each of the three iterators: Fib, UpDown, and SumToN.  It*)
;       (* returns a pointer as the iterator value.  The innermost  *)
;       (* foreach loop uses this pointer to call the appropriate   *)
;       (* iterator.                                                *)
;
;                 foreach iter in MultiIter do
;
;                 (* Okay, this for loop invokes whatever iterator was *)
;                 (* return by the MultiIter iterator above.           *)
;
;                           foreach i in [MultiIter](n) do
;
;                             write(i:3);
;
;                           endfor;
;                           writeln;
;
;                 endfor;
;                 writeln;
;
;       endfor;
;
; end EX12_1;

                  .xlist
                  include   stdlib.a
                  includelibstdlib.lib
                  .list

                  .286                              ;Allow extra adrs modes.


wp                textequ   <word ptr>
ofs               textequ   <offset>


dseg              segment   para public 'code'
dseg              ends


cseg              segment   para public 'code'
                  assume    cs:cseg, ss:sseg


; The following macro builds a resume frame and the returns to the caller
; of an iterator.  It assumes that the iterator and whoever called the
; iterator have the standard activation record defined above and that we
; are building the standard resume frame described above.
;
; This code wipes out the DX register.  Whoever calls the iterator cannot
; count on DX being preserved, likewise, the iterator cannot count on DX
; being preserved across a yield.  Presumably, the iterator returns its
; value in AX.
```

```
Yield           macro
                mov     dx, [BP+2]   ;Place to yield back to.
                push    bp           ;Save Iterator link
                mov     bp, [bp]     ;Get ptr to caller's A.R.
                call    dx           ;Push resume address and rtn.
                pop     bp           ;Restore ptr to our A. R.
                endm


; Fib(n) - Yields the sequence of fibonacci numbers from F(0)..F(n).
;         The fibonacci sequence is defined as:
;
;               F(0) and F(1) = 1.
;               F(n) = F(n-1) + F(n-2) for n > 1.


; The following structure defines the activation record for Fib

CurIndex        textequ  <[bp-6]>    ;Current sequence value.
Fn1             textequ  <[bp-4]>    ;F(n-1) value.
Fn2             textequ  <[bp-2]>    ;F(n-2) value.
DynamicLink     textequ  <[bp]>      ;Saved BP value.
YieldAdrs       textequ  <[bp+2]>    ;Return Adrs for proc.
FailAdrs        textequ  <[bp+4]>    ;Go here when we fail
n               textequ  <[bp+6]>    ;The initial parameter


Fib             proc
                push    bp
                mov     bp, sp
                sub     sp, 6        ;Make room for local variables.

; We will also begin yielding values starting at F(0).
; Since F(0) and F(1) are special cases, yield their values here.

                mov     ax, 1        ;Yield F(0) (we always return at least
                yield                ; F(0)).

                cmp     wp n, 1      ;See if user called this with n=0.
                jb      FailFib
                mov     ax, 1
                yield

; Okay, n >=1 so we need to go into a loop to handle the remaining values.
; First, begin by initializing Fn1 and Fn2 as appropriate.

                mov     wp Fn1, 1
                mov     wp Fn2, 1
                mov     wp CurIndex, 2

WhlLp:          mov     ax, CurIndex ;See if CurIndex > n.
                cmp     ax, n
                ja      FailFib

                push    Fn1
                mov     ax, Fn1
                add     ax, Fn2
                pop     Fn2          ;Fn1 becomes the new Fn2 value.
                mov     Fn1, ax      ;Current value becomes new Fn1 value.
                yield                ;Yield the current value.

                inc     wp CurIndex
                jmp     WhlLp



FailFib:        mov     sp, bp       ;Deallocate local vars.
                pop     bp           ;Restore Dynamic Link.
```

```
                add     sp, 2           ;Skip ret adrs.
                ret     2               ;Return through fail address.
Fib             endp


; UpDown-       This function yields the sequence 0, 1, 2, ..., n, n-1,
;               n-2, ..., 1, 0.

i               textequ <[bp-2]>        ;F(n-2) value.

UpDown          proc
                push    bp
                mov      bp, sp
                sub      sp, 2          ;Make room for i.

                mov      wp i, 0        ;Initialize our index variable (i).
UptoN:          mov      ax, i
                cmp      ax, n
                jae      GoDown

                yield

                inc      wp i
                jmp      UpToN

GoDown:         mov      ax, i
                yield
                mov      ax, i
                cmp      ax, 0
                je       UpDownDone
                dec      wp i
                jmp      GoDown

UpDownDone:     mov      sp, bp         ;Deallocate local vars.
                pop      bp             ;Restore Dynamic Link.
                add      sp, 2          ;Skip ret adrs.
                ret      2              ;Return through fail address.
UpDown          endp



; SumToN(n)-    This iterator returns 1, 2, 3, 6, 10, ... sum(n) where
;               sum(n) = 1+2+3+4+...+n  (e.g., n(n+1)/2);

j               textequ <[bp-2]>
k               textequ  <[bp-4]>

SumToN          proc
                push     bp
                mov      bp, sp
                sub      sp, 4          ;Make room for j and k.

                mov      wp j, 0        ;Initialize our index variable (j).
                mov      wp k, 0        ;Initialize our sum (k).
SumLp:          mov      ax, j
                cmp      ax, n
                ja       SumDone

                add      ax, k
                mov      k, ax

                yield

                inc      wp j
                jmp      SumLp

SumDone:        mov     sp, bp          ;Deallocate local vars.
                pop     bp              ;Restore Dynamic Link.
                add     sp, 2           ;Skip ret adrs.
                ret     2               ;Return through fail address.
```

```
SumToN          endp



; MultiIter- This iterator returns a pointer to each of the above iterators.

MultiIter       proc
                push    bp
                mov     bp, sp

                mov     ax, ofs Fib
                yield
                mov     ax, ofs UpDown
                yield
                mov     ax, ofs SumToN
                yield

                pop     bp
                add     sp, 2
                ret
MultiIter       endp


Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit

; foreach bx in 0..5 do

                mov     bx, 0         ;Loop control variable for outer loop.
WhlBXle5:

; foreach ax in MultiIter do

                push    ofs MultiDone ;Failure address.
                call    MultiIter     ;Get iterator to call.


; foreach i in [ax](bx) do

                push    bx            ;Push "n" (bx) onto the stack.
                push    ofs IterDone ;Failure Address
                call    ax            ;Call the iterator pointed at by the
;                                     ; return value from MultiIter.
;
; write(ax:3);

                mov     cx, 3
                putisize
                ret

; endfor, writeln;

IterDone:       putcr                 ;Writeln;
                ret

; endfor, writeln;

MultiDone:      putcr
                inc     bx
                cmp     bx, 5
                jbe     WhlBXle5

; endfor

Quit:           ExitPgm               ;DOS macro to quit program.
```

```
Main            endp

cseg            ends


sseg            segment  para stack 'stack'
stk             word     1024 dup (0)
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       db       16 dup (?)
zzzzzzseg       ends
                end      Main
```

12.7.2 The 80x86 Enter and Leave Instructions

The following code (Ex12_2.asm on the companion CD-ROM) uses the 80x86 enter and leave instructions to maintain a display in a block structured program. Assemble this program with the "/Zi" option and load it into CodeView. Set breakpoints on the calls to the Lex1, Lex2, Lex3, and Lex4 procedures. Run the program and when you encounter a breakpoint, use the F8 key to single step into each procedure. Single step over the enter instruction (to the following nop). Note the values of the bp and sp register before and after the execution of the enter instruction.

For your lab report: explain the values in the bp and sp registers after executing each enter instruction. Dump memory from ss:sp to about ss:sp+32 using a memory window or the dw command in the command window. Describe the contents of the stack after the execution of each enter instruction.

After executing through the enter instruction in the Lex4 procedure, set a breakpoint on each of the leave instructions. Run the program at full speed (using the F5 key) until you hit each of these leave instructions. Note the values of the bp and sp registers before and after the execution of each leave instruction. **For your lab report:** include these bp/sp values in your lab report and explain them.

```
; EX12_2.asm
;
; Program to demonstrate the ENTER and LEAVE instructions in Chapter 12.
;
; This program simulates the following Pascal code:
;
; program EnterLeave;
; var i:integer;
;
;    procedure Lex1;
;    var j:integer;
;
;      procedure Lex2;
;      var k:integer;
;
;          procedure Lex3;
;          var m:integer;
;
;                procedure Lex4;
;                var n:integer;
;                begin
;
;                         writeln('Lex4');
;                         for i:= 0 to 3 do
;                             for j:= 0 to 2 do
;                           write('(',i,',',j,') ');
;                         writeln;
;                         for k:= 1 downto 0 do
;                           for m:= 1 downto 0 do
;                             for n := 0 to 1 do
;                             write('(',m,',',k,',',n,') ');
```

```
;                         writeln;
;               end;
;
;          begin {Lex3}
;
;               writeln('Lex3');
;               for i := 0 to 1 do
;                   for j := 0 to 1 do
;                        for k := 0 to 1 do
;                            for m := 0 to 1 do
;                          writeln(i,j,k,m);
;
;               Lex4;
;
;          end; {Lex3}
;
;      begin {Lex2}
;
;          writeln('Lex2');
;          for i := 1 downto 0 do
;               for j := 0 to 1 do
;                   for k := 1 downto 0 do
;                        write(i,j,k,' ');
;          writeln;
;
;          Lex3;
;
;      end; {Lex2}
;
;    begin {Lex1}
;
;      writeln('Lex1');
;      Lex2;
;
;    end; {Lex1}
;
; begin {Main (lex0)}
;
;      writeln('Main Program');
;      Lex1;
;
; end.
                .xlist
                include    stdlib.a
                includelib stdlib.lib
                .list

                .286                            ;Allow ENTER & LEAVE.

; Common equates all the procedures use:

wp              texteq  <word ptr>
disp1           texteq  <word ptr [bp-2]>
disp2           texteq  <word ptr [bp-4]>
disp3           texteq  <word ptr [bp-6]>

; Note: the data segment and the stack segment are one and the same in this
; program.  This is done to allow the use of the [bx] addressing mode when
; referencing local and intermediate variables without having to use a
; stack segment prefix.

sseg            segment  para stack 'stack'

i               word     ?                      ;Main program variable.
stk             word     2046 dup (0)

sseg            ends

cseg            segment  para public 'code'
                assume   cs:cseg, ds:sseg, ss:sseg

; Main's activation record looks like this:
;
```

```
;       | return address |<- SP, BP
;       |----------------|

Main            proc
                mov     ax, ss      ;Make SS=DS to simplify addressing
                mov     ds, ax      ; (there will be no need to stick "SS:"
                mov     es, ax      ; in front of addressing modes like
                                    ; "[bx]").

                print
                byte    "Main Program",cr,lf,0
                call    Lex1
Quit:           ExitPgm             ;DOS macro to quit program.
Main            endp

; Lex1's activation record looks like this:
;
;       | return address |
;       |----------------|
;       | Dynamic Link   | <- BP
;       |----------------|
;       | Lex1's AR Ptr  |  | Display
;       |----------------|
;       | J Local var    | <- SP  (BP-4)
;       |----------------|

Lex1_J          textequ  <word ptr [bx-4]>

Lex1            proc    near
                enter   2, 1        ;A 2 byte local variable at lex level 1.
                nop                 ;Spacer instruction for single stepping

                print
                byte    "Lex1",cr,lf,0
                call    Lex2
                leave
                ret
Lex1            endp

; Lex2's activation record looks like this:
;
;       | return address |
;       |----------------|
;       | Dynamic Link   | <- BP
;       |----------------|
;       | Lex1's AR Ptr  |  |
;       |----------------|  | Display
;       | Lex2's AR Ptr  |  |
;       |----------------|
;       | K Local var    | <- SP (BP-6)
;       |----------------|
;
;          writeln('Lex2');
;          for i := 1 downto 0 do
;                for j := 0 to 1 do
;                    for k := 1 downto 0 do
;                         write(i,j,k,' ');
;          writeln;
;
;          Lex3;

Lex2_k          textequ  <word ptr [bx-6]>
k               textequ  <word ptr [bp-6]>

Lex2            proc    near
                enter   2, 2        ;A 2-byte local variable at lex level 2.

                nop                 ;Spacer instruction for single stepping

                print
                byte    "Lex2",cr,lf,0

                mov     i, 1
```

```
ForLpI:         mov     bx, disp1       ;"J" is at lex level one.
                mov     Lex1_J, 0
ForLpJ:         mov     k, 1            ;"K" is local.

ForLpK:         mov     ax, i
                puti
                mov     bx, disp1
                mov     ax, Lex1_J
                puti
                mov     ax, k
                puti
                mov     al, ' '
                putc

                dec     k               ;Decrement from 1->0 and quit
                jns     ForLpK          ; if we hit -1.

                mov     bx, disp1
                inc     Lex1_J
                cmp     Lex1_J, 2
                jb      ForLpJ

                dec     i
                jns     ForLpI

                putcr
                call    Lex3

                leave
                ret
Lex2            endp

; Lex3's activation record looks like this:
;
;       | return address |
;       |----------------|
;       | Dynamic Link   | <- BP
;       |----------------|
;       | Lex1's AR Ptr  |  |
;       |----------------|  |
;       | Lex2's AR Ptr  |  | Display
;       |----------------|  |
;       | Lex3's AR Ptr  |  |
;       |----------------|
;       | M Local var    | <- SP (BP-8)
;       |----------------|
;
;               writeln('Lex3');
;               for i := 0 to 1 do
;                   for j := 0 to 1 do
;                       for k := 0 to 1 do
;                           for m := 0 to 1 do
;                             writeln(i,j,k,m);
;
;               Lex4;

Lex3_M          textequ  <word ptr [bx-8]>
m               textequ  <word ptr [bp-8]>

Lex3            proc    near
                enter   2, 3            ;2-byte variable at lex level 3.

                nop                     ;Spacer instruction for single stepping

                print
                byte    "Lex3",cr,lf,0

                mov     i, 0
ForILp:         mov     bx, disp1
                mov     Lex1_J, 0
ForJlp:         mov     bx, disp2
                mov     Lex2_K, 0
ForKLp:         mov     m, 0
ForMLp:         mov     ax, i
```

```
                        puti
                        mov     bx, disp1
                        mov     ax, Lex1_J
                        puti
                        mov     bx, disp2
                        mov     ax, Lex2_k
                        puti
                        mov     ax, m
                        puti
                        putcr

                        inc     m
                        cmp     m, 2
                        jb      ForMLp

                        mov     bx, disp2
                        inc     Lex2_K
                        cmp     Lex2_K, 2
                        jb      ForKLp

                        mov     bx, disp1
                        inc     Lex1_J
                        cmp     Lex1_J, 2
                        jb      ForJLp

                        inc     i
                        cmp     i, 2
                        jb      ForILp

                        call    Lex4

                        leave
                        ret
Lex3                    endp

; Lex4's activation record looks like this:
;
;       | return address |
;       |----------------|
;       | Dynamic Link   | <- BP
;       |----------------|
;       | Lex1's AR Ptr  |  |
;       |----------------|  |
;       | Lex2's AR Ptr  |  |
;       |----------------|  | Display
;       | Lex3's AR Ptr  |  |
;       |----------------|  |
;       | Lex4's AR Ptr  |  |
;       |----------------|
;       | N Local var    | <- SP (BP-10)
;       |----------------|
;
;
;       writeln('Lex4');
;       for i:= 0 to 3 do
;         for j:= 0 to 2 do
;                 write('(',i,',',j,') ');
;       writeln;
;       for k:= 1 downto 0 do
;         for m:= 1 downto 0 do
;           for n := 0 to 1 do
;                 write('(',m,',',k,',',n,') ');
;       writeln;

n                       textequ  <word ptr [bp-10]>

Lex4                    proc    near
                        enter   2, 4            ;2-byte local variable at lex level 4.

                        nop                     ;Spacer instruction for single stepping

                        print
                        byte    "Lex4",cr,lf,0
```

```
                mov     i, 0
ForILp:         mov     bx, disp1
                mov     Lex1_J, 0
ForJLp:         mov     al, '('
                putc
                mov     ax, i
                puti
                mov     al, ','
                putc
                mov     ax, Lex1_J   ;Note that BX still contains disp1.
                puti
                print
                byte    ") ",0

                inc     Lex1_J       ;BX still contains disp1.
                cmp     Lex1_J, 3
                jb      ForJLp

                inc     i
                cmp     i, 4
                jb      ForILp

                putcr

                mov     bx, disp2
                mov     Lex2_K, 1
ForKLp:         mov     bx, disp3
                mov     Lex3_M, 1
ForMLp:         mov     n, 0
ForNLp:         mov     al, '('
                putc

                mov     bx, disp3
                mov     ax, Lex3_M
                puti
                mov     al, ','
                putc
                mov     bx, disp2
                mov     ax, Lex2_K
                puti
                mov     al, ','
                putc
                mov     ax, n
                puti
                print
                byte    ") ",0

                inc     n
                cmp     n, 2
                jb      ForNLp

                mov     bx, disp3
                dec     Lex3_M
                jns     ForMLp

                mov     bx, disp2
                dec     Lex2_K
                jns     ForKLp

                leave
                ret
Lex4            endp
cseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       db       16 dup (?)
zzzzzzseg       ends
                end      Main
```

12.7.3 Parameter Passing Exercises

The following exercise demonstrates some simple parameter passing. This program passes arrays by reference, word variables by value and by reference, and some functions and procedure by reference. The program itself sorts two arrays using a generic sorting algorithm. The sorting algorithm is generic because the main program passes it a comparison function and a procedure to swap two elements if one is greater than the other.

```
; Ex12_3.asm
;
; This program demonstrates different parameter passing methods.
; It corresponds to the following (pseudo) Pascal code:
;
;
; program main;
; var  i:integer;
;      a:array[0..255] of integer;
;      b:array[0..255] of unsigned;
;
; function LTint(int1, int2:integer):boolean;
; begin
;      LTint := int1 < int2;
; end;
;
; procedure SwapInt(var int1, int2:integer);
; var temp:integer;
; begin
;      temp := int1;
;      int1 := int2;
;      int2 := temp;
; end;
;
; function LTunsigned(uns1, uns2:unsigned):boolean;
; begin
;      LTunsigned := uns1 < uns2;
; end;
;
; procedure SwapUnsigned(uns1, uns2:unsigned);
; var temp:unsigned;
; begin
;      temp := uns1;
;      uns1 := uns2;
;      uns2 := temp;
; end;
;
; (* The following is a simple Bubble sort that will sort arrays containing *)
; (* arbitrary data types.                                                   *)
;
; procedure sort(data:array; elements:integer; function LT:boolean; procedure
swap);
; var i,j:integer;
; begin
;
;      for i := 0 to elements-1 do
;          for j := i+1 to elements do
;                if (LT(data[j], data[i])) then swap(data[i], data[j]);
; end;
;
;
; begin
;
;      for i := 0 to 255 do A[i] := 128-i;
;      for i := 0 to 255 do B[i] := 255-i;
;      sort(A, 256, LTint, SwapInt);
;      sort(B, 256, LTunsigned, SwapUnsigned);
;
;      for i := 0 to 255 do
;      begin
```

```
;                if (i mod 8) = 0 then writeln;
;                write(A[i]:5);
;       end;
;
;       for i := 0 to 255 do
;       begin
;                if (i mod 8) = 0 then writeln;
;                write(B[i]:5);
;       end;
;
; end;

                .xlist
                include     stdlib.a
                includelib  stdlib.lib
                .list

                .386
                option      segment:use16

wp              textequ     <word ptr>

dseg            segment   para public 'data'
A               word      256 dup (?)
B               word      256 dup (?)
dseg            ends

cseg            segment   para public 'code'
                assume    cs:cseg, ds:dseg, ss:sseg

; function LTint(int1, int2:integer):boolean;
; begin
;       LTint := int1 < int2;
; end;
;
; LTint's activation record looks like this:
;
;       |----------------|
;       |      int1      |
;       |----------------|
;       |      int2      |
;       |----------------|
;       | return address |
;       |----------------|
;       |    old BP      |<- SP, BP
;       |----------------|

int1            textequ   <word ptr [bp+6]>
int2            textequ   <word ptr [bp+4]>

LTint           proc      near
                push      bp
                mov       bp, sp

                mov       ax, int1      ;Compare the two parameters
                cmp       ax, int2      ; and return true if int1<int2.
                setl      al            ;Signed comparison here.
                mov       ah, 0         ;Be sure to clear H.O. byte.

                pop       bp
                ret       4
LTint           endp

; Swap's activation record looks like this:
;
;       |----------------|
;       |     Address    |
;       |---    of    ---|
;       |      int1      |
;       |----------------|
;       |     Address    |
;       |---    of    ---|
;       |      int2      |
```

```
;       |----------------|
;       | return address |
;       |----------------|
;       |    old BP      |<- SP, BP
;       |----------------|
;
; The temporary variable is kept in a register.
;
; Note that swapping integers or unsigned integers can be done
; with the same code since the operations are identical for
; either type.
;
; procedure SwapInt(var int1, int2:integer);
; var temp:integer;
; begin
;       temp := int1;
;       int1 := int2;
;       int2 := temp;
; end;
;
; procedure SwapUnsigned(uns1, uns2:unsigned);
; var temp:unsigned;
; begin
;       temp := uns1;
;       uns1 := uns2;
;       uns2 := temp;
; end;
;
int1            textequ  <dword ptr [bp+8]>
int2            textequ  <dword ptr [bp+4]>

SwapInt         proc     near
                push     bp
                mov      bp, sp
                push     es
                push     bx

                les      bx, int1     ;Get address of int1 variable.
                mov      ax, es:[bx]  ;Get int1's value.
                les      bx, int2     ;Get address of int2 variable.
                xchg     ax, es:[bx]  ;Swap int1's value with int2's

                les      bx, int1     ;Get the address of int1 and
                mov      es:[bx], ax  ; store int2's value there.

                pop      bx
                pop      es
                pop      bp
                ret      8
SwapInt         endp

; LTunsigned's activation record looks like this:
;
;       |----------------|
;       |      uns1      |
;       |----------------|
;       |      uns2      |
;       |----------------|
;       | return address |
;       |----------------|
;       |    old BP      |<- SP, BP
;       |----------------|
;
; function LTunsigned(uns1, uns2:unsigned):boolean;
; begin
;       LTunsigned := uns1 < uns2;
; end;

uns1            textequ  <word ptr [bp+6]>
uns2            textequ  <word ptr [bp+4]>

LTunsigned      proc     near
```

```
                push    bp
                mov     bp, sp

                mov     ax, uns1        ;Compare uns1 with uns2 and
                cmp     ax, uns2        ; return true if uns1<uns2.
                setb    al              ;Unsigned comparison.
                mov     ah, 0           ;Return 16-bit boolean.

                pop     bp
                ret     4
LTunsigned      endp

; Sort's activation record looks like this:
;
;       |----------------|
;       |     Data's     |
;       |---           ---|
;       |    Address     |
;       |----------------|
;       |   Elements     |
;       |----------------|
;       |      LT's      |
;       |---           ---|
;       |    Address     |
;       |----------------|
;       |     Swap's     |
;       |---           ---|
;       |    Address     |
;       |----------------|
;       | return address |
;       |----------------|
;       |    old BP      |<- SP, BP
;       |----------------|
;
; procedure sort(data:array; elements:integer; function LT:boolean; procedure
swap);
; var i,j:integer;
; begin
;
;       for i := 0 to elements-1 do
;           for j := i+1 to elements do
;                 if (LT(data[j], data[i])) then swap(data[i], data[j]);
; end;

data            textequ  <dword ptr [bp+10]>
elements        textequ  <word ptr [bp+8]>
funcLT          textequ  <word ptr [bp+6]>
procSwap        textequ  <word ptr [bp+4]>

i               textequ  <word ptr [bp-2]>
j               textequ  <word ptr [bp-4]>

sort            proc     near
                push     bp
                mov      bp, sp
                sub      sp, 4
                push     es
                push     bx

                mov      i, 0
ForILp:         mov      ax, i
                inc      i
                cmp      ax, Elements
                jae      IDone

                mov      j, ax
ForJLp:         mov      ax, j
                cmp      ax, Elements
                ja       JDone

                les      bx, data               ;Push the value of
                mov      si, j                  ; data[j] onto the
                add      si, si                 ; stack.
```

```
                        push    es:[bx+si]

                        les     bx, data                ;Push the value of
                        mov     si, i                   ; data[i] onto the
                        add     si, si                  ; stack.
                        push    es:[bx+si]

                        call    FuncLT                  ;See if data[i] < data[j]
                        cmp     ax, 0                   ;Test boolean result.
                        je      NextJ

                        push    wp data+2               ;Pass data[i] by reference.
                        mov     ax, i
                        add     ax, ax
                        add     ax, wp data
                        push    ax

                        push    wp data+2               ;Pass data[j] by reference.
                        mov     ax, j
                        add     ax, ax
                        add     ax, wp data
                        push    ax

                        call    ProcSwap

NextJ:                  inc     j
                        jmp     ForJLp

JDone:                  inc     i
                        jmp     ForILp

IDone:                  pop     bx
                        pop     es
                        mov     sp, bp
                        pop     bp
                        ret     10
sort                    endp

; Main's activation record looks like this:
;
;       | return address |<- SP, BP
;       |----------------|
;
; begin
;
;       for i := 0 to 255 do A[i] := 128-i;
;       for i := 0 to 255 do B[i] := 33000-i;
;       sort(A, 256, LTint, SwapInt);
;       sort(B, 256, LTunsigned, SwapUnsigned);
;
;       for i := 0 to 255 do
;       begin
;                   if (i mod 8) = 0 then writeln;
;                   write(A[i]:5);
;       end;
;
;       for i := 0 to 255 do
;       begin
;                   if (i mod 8) = 0 then writeln;
;                   write(B[i]:5);
;       end;
;
; end;

Main                    proc
                        mov     ax, dseg    ;Initialize the segment registers.
                        mov     ds, ax
                        mov     es, ax

; Note that the following code merges the two initialization for loops
; into a single loop.

                        mov     ax, 128
                        mov     bx, 0
```

```
                mov     cx, 33000
ForILp:         mov     A[bx], ax
                mov     B[bx], cx
                add     bx, 2
                dec     ax
                dec     cx
                cmp     bx, 256*2
                jb      ForILp

                push    ds                      ;Seg address of A
                push    offset A                ;Offset of A
                push    256                     ;# of elements in A
                push    offset LTint            ;Address of compare routine
                push    offset SwapInt          ;Address of swap routine
                call    Sort

                push    ds                      ;Seg address of B
                push    offset B                ;Offset of B
                push    256                     ;# of elements in A
                push    offset LTunsigned       ;Address of compare routine
                push    offset SwapInt          ;Address of swap routine
                call    Sort

; Print the values in A.

                mov     bx, 0
ForILp2:        test    bx, 0Fh                 ;See if (I mod 8) = 0
                jnz     NotMod                  ; note: BX mod 16 = I mod 8.
                putcr
NotMod:         mov     ax, A[bx]
                mov     cx, 5
                putisize
                add     bx, 2
                cmp     bx, 256*2
                jb      ForILp2

; Print the values in B.
                mov     bx, 0
ForILp3:        test    bx, 0Fh                 ;See if (I mod 8) = 0
                jnz     NotMod2                 ; note: BX mod 16 = I mod 8.
                putcr
NotMod2:        mov     ax, B[bx]
                mov     cx, 5
                putusize
                add     bx, 2
                cmp     bx, 256*2
                jb      ForILp3

Quit:           ExitPgm                 ;DOS macro to quit program.
Main            endp
cseg            ends

sseg            segment  para stack 'stack'
stk             word     256 dup (0)
sseg            ends

zzzzzzseg       segment  para public 'zzzzzz'
LastBytes       db       16 dup (?)
zzzzzzseg       ends
                end      Main
```

12.8 Programming Projects

1) Write at iterator to which you pass an array of characters by reference. The iterator should return an index into the array that points at a whitespace character (any ASCII code less than or equal to a space) it finds. On each call, the iterator should return the index of the next whitespace character. The iterator should fail if it encounters a byte containing the value zero. Use local variables for any values the iterator needs.

2) Write a recursive routine that does the following:

```
function recursive(i:integer):integer;
var j,k:integer;
begin
          j := i;
          k := i*i;
          if (i >= 0) then writeln('AR Address =', Recursive(i-1));
          writeln(i,' ',j,' ',k);
          recursive := Value in BP Register;
end;
```

From your main program, call this procedure and pass it the value 10 on the stack. Verify that you get correct results back. Explain the results.

3) Write a program that contains a procedure to which you pass four parameters on the stack. These should be passed by value, reference, value-result, and result, respectively (for the value-result parameter, pass the address of the object on the stack). Inside that procedure, you should call three other procedures that also take four parameters (each). However, the first parameter should use pass by value for all four parameters; the second procedure should use pass by reference for all four parameters; and the third should use pass by value-result for all four parameters. Pass the four parameters in the enclosing procedure as parameters to each of these three child procedures. Inside the child procedures, print the parameter's values and change their results. Immediately upon return from each of these child procedures, print the parameters' values. Write a main program that passes four local (to the main program) variables you've initialized with different values to the first procedure above. Run the program and verify that it is operating correctly and that it is passing the parameters to each of these procedures in a reasonable fashion.

4) Write a program that implements the following Pascal program in assembly language. Assume that all program variables (including globals in the main program) are allocated in activation records on the stack.

```
program nest3;
var    i:integer;

       procedure A(k:integer);

                 procedure B(procedure c);
                 var m:integer;
                 begin

                          for m:= 0 to 4 do c(m);

                 end; {B}

                 procedure D(n:integer);
                 begin

                          for i:= 0 to n-1 do writeln(i);

                 end; {D}

                 procedure E;
                 begin

                          writeln('A stuff:');
                          B(A);
                          writeln('D stuff:');
                          B(D);

                 end; {E}

       begin {A}

                 B(D);
                 writeln;
                 if k < 2 then E;
```

```
        end; {A}

begin {nest3}

        A(0);

end; {nest3}
```

5) The program in Section 12.7.2 (Ex12_2.asm on the companion CD-ROM) uses the 80286 enter and leave instructions to maintain the display in each activation record. As pointed out in Section 12.1.6, these instructions are quite slow, especially on 80486 and later processors. Rewrite this code by replacing the enter and leave instructions with the straight-line code that does the same job. In CodeView, single step through the program as per the second laboratory exercise (Section 12.7.2) to verify that your stack frames are identical to those the enter and leave instructions produce.

6) The generic Bubble Sort program in Section 12.7.3 only works with data objects that are two bytes wide. This is because the Sort procedure passes the values of Data[I] and Data[J] on the stack to the comparison routines (LTint and LTunsigned) and because the sort routine multiplies the i and j indexes by two when indexing into the data array. This is a severe shortcoming to this generic sort routine. Rewrite the program to make it truly generic. Do this by writing a "CompareAndSwap" routine that will replace the LT and Swap calls. To CompareAndSwap you should pass the array (by reference) and the two array indexes (i and j) to compare and possibly swap. Write two versions of the CompareAndSwap routine, one for unsigned integers and one for signed integers. Run this program and verify that your implementation works properly.

12.9 Summary

Block structured languages, like Pascal, provide access to non-local variables at different lex levels. Accessing non-local variables is a complex task requiring special data structures such as a static link chain or a display. The display is probably the most efficient way to access non-local variables. The 80286 and later processors provide special instructions, enter and leave for maintaining a display list, but these instructions are too slow for most common uses. For additional details, see

- "Lexical Nesting, Static Links, and Displays" on page 639
- "Scope" on page 640
- "Static Links" on page 642
- "Accessing Non-Local Variables Using Static Links" on page 647
- "The Display" on page 648
- "The 80286 ENTER and LEAVE Instructions" on page 650
- "Passing Variables at Different Lex Levels as Parameters." on page 652
- "Passing Parameters as Parameters to Another Procedure" on page 655
- "Passing Procedures as Parameters" on page 659

Iterators are a cross between a function and a looping construct. They are a very powerful programming construct available in many very high level languages. Efficient implementation of iterators involves careful manipulation of the stack at run time. To see how to implement iterators, read the following sections:

- "Iterators" on page 663
- "Implementing Iterators Using In-Line Expansion" on page 664
- "Implementing Iterators with Resume Frames" on page 666
- "An Example of an Iterator" on page 669
- "Another Iterator Example" on page 673

12.10 Questions

1) What is an iterator?

2) What is a resume frame?

3) How do the iterators in this chapter implement the success and failure results?

4) What does the stack look like when executing the body of a loop controlled by an iterator?

5 What is a static link?

6) What is a display?

7) Describe how to access a non-local variable when using static links.

8) Describe how to access a non-local variable when using a display.

9) How would you access a non-local variable when using the display built by the 80286 ENTER instruction?

10) Draw a picture of the activation record for a procedure at lex level 4 that uses the ENTER instruction to build the display.

11) Explain why the static links work better than a display when passing procedures and functions as parameters.

12) Suppose you want to pass an intermediate variable by value-result using the technique where you push the value before calling the procedure and then pop the value (storing it back into the intermediate variable) upon return from the procedure. Provide two examples, one using static links and one using a display, that implement pass by value-result in this fashion.

13) Convert the following (pseudo) Pascal code into 80x86 assembly language. Assume Pascal supports pass by name and pass by lazy evaluation parameters as suggested by the following code.

```
program main;
var k:integer;

procedure one(LazyEval i:integer);
begin
       writeln(i);
end;

procedure two(name j:integer);
begin
       one(j);
end;

begin {main}
       k := 2;
       two(k);
end;
```