Procedures and Functions — Chapter 11

Modular design is one of the cornerstones of structured programming. A modular program contains blocks of code with single entry and exit points. You can *reuse* well written sections of code in other programs or in other sections of an existing program. If you reuse an existing segment of code, you needn't design, code, nor debug that section of code since (presumably) you've already done so. Given the rising costs of software development, modular design will become more important as time passes.

The basic unit of a modular program is the module. Modules have different meanings to different people, herein you can assume that the terms module, subprogram, subroutine, program unit, procedure, and function are all synonymous.

The procedure is the basis for a programming style. The procedural languages include Pascal, BASIC, C++, FORTRAN, PL/I, and ALGOL. Examples of non-procedural languages include APL, LISP, SNOBOL4 ICON, FORTH, SETL, PROLOG, and others that are based on other programming constructs such as functional abstraction or pattern matching. Assembly language is capable of acting as a procedural or non-procedural language. Since you're probably much more familiar with the procedural programming paradigm this text will stick to simulating procedural constructs in 80x86 assembly language.

11.0 Chapter Overview

This chapter presents an introduction to procedures and functions in assembly language. It discusses basic principles, parameter passing, function results, local variables, and recursion. You will use most of the techniques this chapter discusses in typical assembly language programs. The discussion of procedures and functions continues in the next chapter; that chapter discusses advanced techniques that you will not commonly use in assembly language programs. The sections below that have a "•" prefix are essential. Those sections with a "❏" discuss advanced topics that you may want to put off for a while.

- • Procedures.
- ❏ Near and far procedures.
- • Functions
- • Saving the state of the machine
- • Parameters
- • Pass by value parameters.
- • Pass by reference parameters.
- ❏ Pass by value-returned parameters.
- ❏ Pass by result parameters.
- ❏ Pass by name parameters.
- • Passing parameters in registers.
- • Passing parameters in global variables.
- • Passing parameters on the stack.
- • Passing parameters in the code stream.
- ❏ Passing parameters via a parameter block.
- • Function results.
- • Returning function results in a register.
- • Returning function results on the stack.
- • Returning function results in memory locations.
- • Side effects.
- ❏ Local variable storage.
- ❏ Recursion.

11.1 Procedures

In a procedural environment, the basic unit of code is the *procedure*. A procedure is a set of instructions that compute some value or take some action (such as printing or reading a character value). The definition of a procedure is very similar to the definition of an *algorithm*. A procedure is a set of rules to follow which, if they conclude, produce some result. An algorithm is also such a sequence, but an algorithm is guaranteed to terminate whereas a procedure offers no such guarantee.

Most procedural programming languages implement procedures using the call/return mechanism. That is, some code calls a procedure, the procedure does its thing, and then the procedure returns to the caller. The call and return instructions provide the 80x86's *procedure invocation mechanism*. The calling code calls a procedure with the call instruction, the procedure returns to the caller with the ret instruction. For example, the following 80x86 instruction calls the UCR Standard Library sl_putcr routine[1]:

```
                call    sl_putcr
```

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

Alas, the UCR Standard Library does not supply all the routines you will need. Most of the time you'll have to write your own procedures. A simple procedure may consist of nothing more than a sequence of instructions ending with a ret instruction. For example, the following "procedure" zeros out the 256 bytes starting at the address in the bx register:

```
ZeroBytes:      xor     ax, ax
                mov     cx, 128
ZeroLoop:       mov     [bx], ax
                add     bx, 2
                loop    ZeroLoop
                ret
```

By loading the bx register with the address of some block of 256 bytes and issuing a call ZeroBytes instruction, you can zero out the specified block.

As a general rule, you won't define your own procedures in this manner. Instead, you should use MASM's proc and endp assembler directives. The ZeroBytes routine, using the proc and endp directives, is

```
ZeroBytes       proc
                xor     ax, ax
                mov     cx, 128
ZeroLoop:       mov     [bx], ax
                add     bx, 2
                loop    ZeroLoop
                ret
ZeroBytes       endp
```

Keep in mind that proc and endp are assembler directives. They do not generate any code. They're simply a mechanism to help make your programs easier to read. To the 80x86, the last two examples are identical; however, to a human being, latter is clearly a self-contained procedure, the other could simply be an arbitrary set of instructions within some other procedure. Consider now the following code:

```
ZeroBytes:      xor     ax, ax
                jcxz    DoFFs
ZeroLoop:       mov     [bx], ax
                add     bx, 2
                loop    ZeroLoop
                ret

DoFFs:          mov     cx, 128
                mov     ax, 0ffffh
```

1. Normally you would use the putcr macro to accomplish this, but this call instruction will accomplish the same thing.

```
FFLoop:         mov     [bx], ax
                sub     bx, 2
                loop    FFLoop
                ret
```

Are there two procedures here or just one? In other words, can a calling program enter this code at labels ZeroBytes and DoFFs or just at ZeroBytes? The use of the proc and endp directives can help remove this ambiguity:

Treated as a single subroutine:

```
ZeroBytes       proc
                xor     ax, ax
                jcxz    DoFFs
ZeroLoop:       mov     [bx], ax
                add     bx, 2
                loop    ZeroLoop
                ret

DoFFs:          mov     cx, 128
                mov     ax, 0ffffh
FFLoop:         mov     [bx], ax
                sub     bx, 2
                loop    FFLoop
                ret
ZeroBytes       endp
```

Treated as two separate routines:

```
ZeroBytes       proc
                xor     ax, ax
                jcxz    DoFFs
ZeroLoop:       mov     [bx], ax
                add     bx, 2
                loop    ZeroLoop
                ret
ZeroBytes       endp

DoFFs           proc
                mov     cx, 128
                mov     ax, 0ffffh
FFLoop:         mov     [bx], ax
                sub     bx, 2
                loop    FFLoop
                ret
DoFFs           endp
```

Always keep in mind that the proc and endp directives are *logical* procedure separators. The 80x86 microprocessor returns from a procedure by executing a ret instruction, not by encountering an endp directive. The following is not equivalent to the code above:

```
ZeroBytes       proc
                xor     ax, ax
                jcxz    DoFFs
ZeroLoop:       mov     [bx], ax
                add     bx, 2
                loop    ZeroLoop
;       Note missing RET instr.
ZeroBytes       endp

DoFFs           proc
                mov     cx, 128
                mov     ax, 0ffffh
FFLoop:         mov     [bx], ax
                sub     bx, 2
                loop    FFLoop
;       Note missing RET instr.
DoFFs           endp
```

Without the ret instruction at the end of each procedure, the 80x86 will fall into the next subroutine rather than return to the caller. After executing ZeroBytes above, the 80x86 will drop through to the DoFFs subroutine (beginning with the mov cx, 128 instruction).

Once DoFFs is through, the 80x86 will continue execution with the next executable instruction following DoFFs' endp directive.

An 80x86 procedure takes the form:

```
ProcName        proc      {near|far}   ;Choose near, far, or neither.
       <Procedure instructions>
ProcName        endp
```

The near or far operand is optional, the next section will discuss its purpose. The procedure name must be on the both proc and endp lines. The procedure name must be unique in the program.

Every proc directive must have a matching endp directive. Failure to match the proc and endp directives will produce a *block nesting error*.

11.2 Near and Far Procedures

The 80x86 supports near and far subroutines. Near calls and returns transfer control between procedures in the same code segment. Far calls and returns pass control between different segments. The two calling and return mechanisms push and pop different return addresses. You generally do not use a near call instruction to call a far procedure or a far call instruction to call a near procedure. Given this little rule, the next question is "how do you control the emission of a near or far call or ret?"

Most of the time, the call instruction uses the following syntax:

```
                call      ProcName
```

and the ret instruction is either[2]:

```
                ret
or              ret       disp
```

Unfortunately, these instructions do not tell MASM if you are calling a near or far procedure or if you are returning from a near or far procedure. The proc directive handles that chore. The proc directive has an optional operand that is either near or far. Near is the default if the operand field is empty[3]. The assembler assigns the procedure type (near or far) to the symbol. Whenever MASM assembles a call instruction, it emits a near or far call depending on operand. Therefore, declaring a symbol with proc or proc near, forces a near call. Likewise, using proc far, forces a far call.

Besides controlling the generation of a near or far call, proc's operand also controls ret code generation. If a procedure has the near operand, then all return instructions inside that procedure will be near. MASM emits far returns inside far procedures.

11.2.1 Forcing NEAR or FAR CALLs and Returns

Once in a while you might want to override the near/far declaration mechanism. MASM provides a mechanism that allows you to force the use of near/far calls and returns.

Use the near ptr and far ptr operators to override the automatic assignment of a near or far call. If NearLbl is a near label and FarLbl is a far label, then the following call instructions generate a near and far call, respectively:

```
                call      NearLbl                 ;Generates a NEAR call.
                call      FarLbl                  ;Generates a FAR call.
```

Suppose you need to make a far call to NearLbl or a near call to FarLbl. You can accomplish this using the following instructions:

2. There are also retn and retf instructions.
3. Unless you are using MASM's *simplified segment directives*. See the appendices for details.

```
                call    far ptr NearLbl         ;Generates a FAR call.
                call    near ptr FarLbl         ;Generates a NEAR call.
```

Calling a near procedure using a far call, or calling a far procedure using a near call isn't something you'll normally do. If you call a near procedure using a far call instruction, the near return will leave the cs value on the stack. Generally, rather than:

```
                call    far ptr NearProc
```

you should probably use the clearer code:

```
                push    cs
                call    NearProc
```

Calling a far procedure with a near call is a very dangerous operation. If you attempt such a call, the current cs value must be on the stack. Remember, a far ret pops a segmented return address off the stack. A near call instruction only pushes the offset, not the segment portion of the return address.

Starting with MASM v5.0, there are explicit instructions you can use to force a near or far ret. If ret appears within a procedure declared via proc and end;, MASM will automatically generate the appropriate near or far return instruction. To accomplish this, use the retn and retf instructions. These two instructions generate a near and far ret, respectively.

11.2.2 Nested Procedures

MASM allows you to nest procedures. That is, one procedure definition may be totally enclosed inside another. The following is an example of such a pair of procedures:

```
OutsideProc     proc    near
                jmp     EndofOutside

InsideProc      proc    near
                mov     ax, 0
                ret
InsideProc      endp

EndofOutside:   call    InsideProc
                mov     bx, 0
                ret
OutsideProc     endp
```

Unlike some high level languages, nesting procedures in 80x86 assembly language doesn't serve any useful purpose. If you nest a procedure (as with InsideProc above), you'll have to code an explicit jmp around the nested procedure. Placing the nested procedure after all the code in the outside procedure (but still between the outside proc/endp directives) doesn't accomplish anything. Therefore, there isn't a good reason to nest procedures in this manner.

Whenever you nest one procedure within another, it must be totally contained within the nesting procedure. That is, the proc and endp statements for the nested procedure must lie between the proc and endp directives of the outside, nesting, procedure. The following is *not* legal:

```
OutsideProc     proc    near
                  .
                  .
                  .
InsideProc      proc    near
                  .
                  .
                  .
OutsideProc     endp
                  .
                  .
                  .
InsideProc      endp
```

The OutsideProc and InsideProc procedures overlap, they are not nested. If you attempt to create a set of procedures like this, MASM would report a "block nesting error". Figure 11.1 demonstrates this graphically.

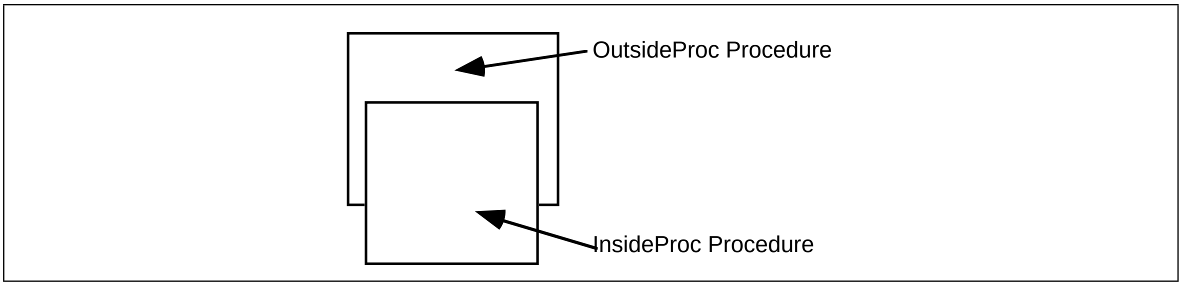

Figure 11.1 Illegal Procedure Nesting

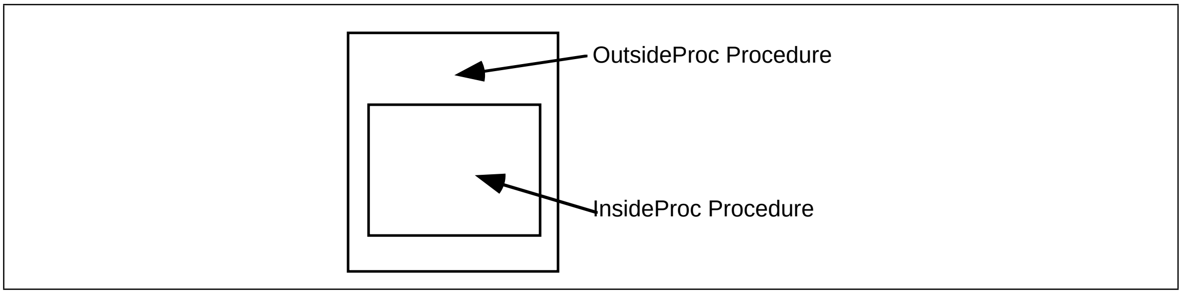

Figure 11.2 Legal Procedure Nesting

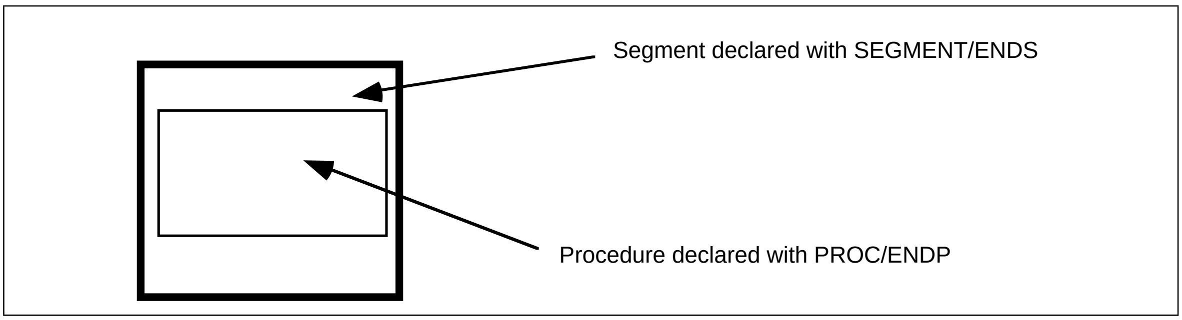

Figure 11.3 Legal Procedure/Segment Nesting

The only form acceptable to MASM appears in Figure 11.2.

Besides fitting inside an enclosing procedure, proc/endp groups must fit entirely within a segment. Therefore the following code is illegal:

```
cseg            segment
MyProc          proc     near
                ret
cseg            ends
MyProc          endp
```

The endp directive must appear before the cseg ends statement since MyProc begins inside cseg. Therefore, procedures within segments must always take the form shown in Figure 11.3.

Not only can you nest procedures inside other procedures and segments, but you can nest segments inside other procedures and segments as well. If you're the type who likes to simulate Pascal or C procedures in assembly language, you can create variable declaration sections at the beginning of each procedure you create, just like Pascal:

```
cgroup          group    cseg1, cseg2

cseg1           segment  para public 'code'
cseg1           ends

cseg2           segment    para public 'code'
cseg2           ends
```

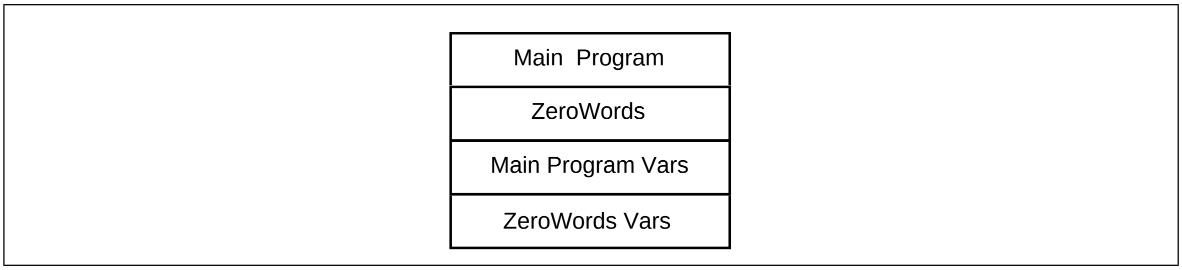

Figure 11.4 Example Memory Layout

```
dseg            segment    para public 'data'
dseg            ends

cseg1           segment    para public 'code'
                assume     cs:cgroup, ds:dseg

MainPgm         proc       near

; Data declarations for main program:

dseg            segment    para public 'data'
I               word       ?
J               word       ?
dseg            ends

; Procedures that are local to the main program:

cseg2           segment    para public 'code'

ZeroWords       proc       near

; Variables local to ZeroBytes:

dseg            segment    para public 'data'
AXSave          word       ?
BXSave          word       ?
CXSave          word       ?
dseg            ends

; Code for the ZeroBytes procedure:

                mov        AXSave, ax
                mov        CXSave, cx
                mov        BXSave, bx
                xor        ax, ax
ZeroLoop:       mov        [bx], ax
                inc        bx
                inc        bx
                loop       ZeroLoop
                mov        ax, AXSave
                mov        bx, BXSave
                mov        cx, CXSave
                ret
ZeroWords       endp

Cseg2           ends

; The actual main program begins here:

                mov        bx, offset Array
                mov        cx, 128
                call       ZeroWords
                ret
MainPgm         endp
cseg1           ends
                end
```

The system will load this code into memory as shown in Figure 11.4.

ZeroWords *follows* the main program because it belongs to a different segment (cseg2) than MainPgm (cseg1). Remember, the assembler and linker combine segments with the same class name into a single segment before loading them into memory (see “Segment Loading Order” on page 368 for more details). You can use this feature of the assembler to “pseudo-Pascalize” your code in the fashion shown above. However, you’ll probably not find your programs to be any more readable than using the straight forward non-nesting approach.

11.3 Functions

The difference between functions and procedures in assembly language is mainly a matter of definition. The purpose for a function is to return some explicit value while the purpose for a procedure is to execute some action. To declare a function in assembly language, use the proc/endp directives. All the rules and techniques that apply to procedures apply to functions. This text will take another look at functions later in this chapter in the section on function results. From here on, procedure will mean procedure or function.

11.4 Saving the State of the Machine

Take a look at this code:

```
                mov     cx, 10
Loop0:          call    PrintSpaces
                putcr
                loop    Loop0
                  .
                  .
                  .
PrintSpaces     proc    near
                mov     al, ' '
                mov     cx, 40
PSLoop:         putc
                loop    PSLoop
                ret
PrintSpaces     endp
```

This section of code attempts to print ten lines of 40 spaces each. Unfortunately, there is a subtle bug that causes it to print 40 spaces per line in an infinite loop. The main program uses the loop instruction to call PrintSpaces 10 times. PrintSpaces uses cx to count off the 40 spaces it prints. PrintSpaces returns with cx containing zero. The main program then prints a carriage return/line feed, decrements cx, and then repeats because cx isn’t zero (it will always contain 0FFFFh at this point).

The problem here is that the PrintSpaces subroutine doesn’t preserve the cx register. Preserving a register means you save it upon entry into the subroutine and restore it before leaving. Had the PrintSpaces subroutine preserved the contents of the cx register, the program above would have functioned properly.

Use the 80x86’s push and pop instructions to preserve register values while you need to use them for something else. Consider the following code for PrintSpaces:

```
PrintSpaces     proc    near
                push    ax
                push    cx
                mov     al, ' '
                mov     cx, 40
PSLoop:         putc
                loop    PSLoop
                pop     cx
                pop     ax
                ret
PrintSpaces     endp
```

Note that PrintSpaces saves and restores ax and cx (since this procedure modifies these registers). Also, note that this code pops the registers off the stack in the reverse order that it pushed them. The operation of the stack imposes this ordering.

Either the caller (the code containing the call instruction) or the callee (the subroutine) can take responsibility for preserving the registers. In the example above, the callee preserved the registers. The following example shows what this code might look like if the caller preserves the registers:

```
                mov     cx, 10
Loop0:          push    ax
                push    cx
                call    PrintSpaces
                pop     cx
                pop     ax
                putcr
                loop    Loop0
                  .
                  .
                  .
PrintSpaces     proc    near
                mov     al, ' '
                mov     cx, 40
PSLoop:         putc
                loop    PSLoop
                ret
PrintSpaces     endp
```

There are two advantages to callee preservation: space and maintainability. If the callee preserves all affected registers, then there is only one copy of the push and pop instructions, those the procedure contains. If the caller saves the values in the registers, the program needs a set of push and pop instructions around every call. Not only does this make your programs longer, it also makes them harder to maintain. Remembering which registers to push and pop on each procedure call is not something easily done.

On the other hand, a subroutine may unnecessarily preserve some registers if it preserves all the registers it modifies. In the examples above, the code needn't save ax. Although PrintSpaces changes the al, this won't affect the program's operation. If the caller is preserving the registers, it doesn't have to save registers it doesn't care about:

```
                mov     cx, 10
Loop0:          push    cx
                call    PrintSpaces
                pop     cx
                putcr
                loop    Loop0
                putcr
                putcr
                call    PrintSpaces

                mov     al, '*'
                mov     cx, 100
Loop1:          putc
                push    ax
                push    cx
                call    PrintSpaces
                pop     cx
                pop     ax
                putc
                putcr
                loop    Loop1
                  .
                  .
                  .
PrintSpaces     proc    near
                mov     al, ' '
                mov     cx, 40
PSLoop:         putc
                loop    PSLoop
                ret
PrintSpaces     endp
```

This example provides three different cases. The first loop (Loop0) only preserves the cx register. Modifying the al register won't affect the operation of this program. Immediately after the first loop, this code calls PrintSpaces again. However, this code doesn't save ax or cx because it doesn't care if PrintSpaces changes them. Since the final loop (Loop1) uses ax and cx, it saves them both.

One big problem with having the caller preserve registers is that your program may change. You may modify the calling code or the procedure so that they use additional registers. Such changes, of course, may change the set of registers that you must preserve. Worse still, if the modification is in the subroutine itself, you will need to locate *every* call to the routine and verify that the subroutine does not change any registers the calling code uses.

Preserving registers isn't all there is to preserving the environment. You can also push and pop variables and other values that a subroutine might change. Since the 80x86 allows you to push and pop memory locations, you can easily preserve these values as well.

11.5 Parameters

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

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

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

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

You also have to worry about where you can pass parameters. Common places are

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

Finally, the amount of data has a direct bearing on where and how to pass it. The following sections take up these issues.

11.5.1 Pass by Value

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

```
CallProc(I);
```

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

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

11.5.2 Pass by Reference

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

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

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

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

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

                end;

                  .
                  .
                  .

       begin {main}

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

       end.
```

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

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

11.5.3 Pass by Value-Returned

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

The Pascal code presented in the previous section would operate properly with pass by value-returned parameters. Of course, when Bletch returns to the calling code, m could only contain one of the two values, but while Bletch is executing, i and j would contain distinct values.

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

11.5.4 Pass by Result

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

11.5.5 Pass by Name

Pass by name is the parameter passing mechanism used by macros, text equates, and the #define macro facility in the C programming language. This parameter passing mechanism uses textual substitution on the parameters. Consider the following MASM macro:

```
PassByName          macro     Parameter1, Parameter2
                    mov       ax, Parameter1
                    add       ax, Parameter2
                    endm
```

If you have a macro invocation of the form:

```
                    PassByName  bx, I
```

MASM emits the following code, substituting bx for Parameter1 and I for Parameter2:

```
                    mov       ax, bx
                    add       ax, I
```

Some high level languages, such as ALGOL-68 and Panacea, support pass by name parameters. However, implementing pass by name using textual substitution in a compiled language (like ALGOL-68) is very difficult and inefficient. Basically, you would have to recompile a function everytime you call it. So compiled languages that support pass by name parameters generally use a different technique to pass those parameters. Consider the following Panacea procedure:

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

       foreach index in 0..10 do

                 item := 0;

       endfor;

end PassByName;
```

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

```
begin PassByName;

       foreach index in 0..10 do

                 A[I] := 0; (* Note that index and I are aliases *)

       endfor;

end PassByName;
```

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

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

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

```
ItemThunk       proc    near
                mov     bx, I
                shl     bx, 1
                lea     bx, A[bx]
                ret
ItemThunk       endp
```

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

```
; item := 0;

                call    ItemThunk
                mov     word ptr [bx], 0
```

Thunk is the historical term for these functions that compute the address of a pass by name parameter. It is worth noting that most HLLs supporting pass by name parameters do not call thunks directly (like the call above). Generally, the caller passes the address of a thunk and the subroutine calls the thunk *indirectly*. This allows the same sequence of instructions to call several different thunks (corresponding to different calls to the subroutine).

11.5.6 Pass by Lazy-Evaluation

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

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

```
PassByEval: procedure(eval a:integer; eval b:integer; eval c:integer);
```

When you call the PassByEval function it does not evaluate the actual parameters and pass their values to the procedure. Instead, the compiler generates thunks that will compute the value of the parameter at most one time. If the first access to an eval parameter is a read, the thunk will compute the parameter's value and store that into a local variable. It will also set a flag so that all future accesses will not call the thunk (since it has already computed the parameter's value). If the first access to an eval parameter is a write, then the code sets the flag and future accesses within the same procedure activation will use the written value and ignore the thunk.

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

11.5.7 Passing Parameters in Registers

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

Data Size	Pass in this Register
Byte:	al
Word:	ax
Double Word:	dx:ax or eax (if 80386 or better)

This is, by no means, a hard and fast rule. If you find it more convenient to pass 16 bit values in the si or bx register, by all means do so. However, most programmers use the registers above to pass parameters.

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

First Last

ax, dx, si, di, bx, cx

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

The UCR Standard Library package provides several good examples of procedures that pass parameters by value in the registers. Putc, which outputs an ASCII character code to the video display, expects an ASCII value in the al register. Likewise, puti expects the value of a signed integer in the ax register. As another example, consider the following putsi (put short integer) routine that outputs the value in al as a signed integer:

```
putsi           proc
                push    ax                      ;Save AH's value.
                cbw                             ;Sign extend AL -> AX.
                puti                            ;Let puti do the real work.
                pop     ax                      ;Restore AH.
                ret
putsi           endp
```

The other four parameter passing mechanisms (pass by reference, value-returned, result, and name) generally require that you pass a pointer to the desired object (or to a thunk in the case of pass by name). When passing such parameters in registers, you have to consider whether you're passing an offset or a full segmented address. Sixteen bit offsets can be passed in any of the 80x86's general purpose 16 bit registers. si, di, and bx are the best place to pass an offset since you'll probably need to load it into one of these registers anyway[4]. You can pass 32 bit segmented addresses dx:ax like other double word parameters. However, you can also pass them in ds:bx, ds:si, ds:di, es:bx, es:si, or es:di and be able to use them without copying into a segment register.

The UCR Stdlib routine puts, which prints a string to the video display, is a good example of a subroutine that uses pass by reference. It wants the address of a string in the es:di register pair. It passes the parameter in this fashion, not because it modifies the parameter, but because strings are rather long and passing them some other way would be inefficient. As another example, consider the following strfill(str,c) that copies the char-

4. This does not apply to thunks. You may pass the address of a thunk in any 16 bit register. Of course, on an 80386 or later processor, you can use any of the 80386's 32-bit registers to hold an address.

acter c (passed by value in al) to each character position in str (passed by reference in es:di) up to a zero terminating byte:

```
; strfill-       copies value in al to the string pointed at by es:di
;                up to a zero terminating byte.

byp              textequ  <byte ptr>

strfill          proc
                 pushf                            ;Save direction flag.
                 cld                              ;To increment D with STOS.
                 push     di                      ;Save, because it's changed.
                 jmp      sfStart

sfLoop:          stosb                            ;es:[di] := al, di := di + 1;
sfStart:         cmp      byp es:[di], 0          ;Done yet?
                 jne      sfLoop

                 pop      di                      ;Restore di.
                 popf                             ;Restore direction flag.
                 ret
strfill          endp
```

When passing parameters by value-returned or by result to a subroutine, you could pass in the address in a register. Inside the procedure you would copy the value pointed at by this register to a local variable (value-returned only). Just before the procedure returns to the caller, it could store the final result back to the address in the register.

The following code requires two parameters. The first is a pass by value-returned parameter and the subroutine expects the address of the actual parameter in bx. The second is a pass by result parameter whose address is in si. This routine increments the pass by value-result parameter and stores the previous result in the pass by result parameter:

```
; CopyAndInc-    BX contains the address of a variable. This routine
;                copies that variable to the location specified in SI
;                and then increments the variable BX points at.
;                Note: AX and CX hold the local copies of these
;                parameters during execution.

CopyAndInc       proc
                 push     ax          ;Preserve AX across call.
                 push     cx          ;Preserve CX across call.
                 mov      ax, [bx]    ;Get local copy of 1st parameter.
                 mov      cx, ax      ;Store into 2nd parm's local var.
                 inc      ax          ;Increment 1st parameter.
                 mov      [si], cx    ;Store away pass by result parm.
                 mov      [bx], ax    ;Store away pass by value/ret parm.
                 pop      cx          ;Restore CX's value.
                 pop      ax          ;Restore AX's value.
                 ret
CopyAndInc       endp
```

To make the call CopyAndInc(I,J) you would use code like the following:

```
                 lea      bx, I
                 lea      si, J
                 call     CopyAndInc
```

This is, of course, a trivial example whose implementation is very inefficient. Nevertheless, it shows how to pass value-returned and result parameters in the 80x86's registers. If you are willing to trade a little space for some speed, there is another way to achieve the same results as pass by value-returned or pass by result when passing parameters in registers. Consider the following implementation of CopyAndInc:

```
CopyAndInc       proc
                 mov      cx, ax      ;Make a copy of the 1st parameter,
                 inc      ax          ; then increment it by one.
                 ret
CopyAndInc       endp
```

To make the CopyAndInc(I,J) call, as before, you would use the following 80x86 code:

```
                mov     ax, I
                call    CopyAndInc
                mov     I, ax
                mov     J, cx
```

Note that this code does not pass any addresses at all; yet it has the same semantics (that is, performs the same operations) as the previous version. Both versions increment I and store the pre-incremented version into J. Clearly the latter version is faster, although your program will be slightly larger if there are many calls to CopyAndInc in your program (six or more).

You can pass a parameter by name or by lazy evaluation in a register by simply loading that register with the address of the thunk to call. Consider the Panacea PassByName procedure (see "Pass by Name" on page 576). One implementation of this procedure could be the following:

```
;PassByName-    Expects a pass by reference parameter index
;               passed in si and a pass by name parameter, item,
;               passed in dx (the thunk returns the address in bx).

PassByName      proc
                push    ax                      ;Preserve AX across call
                mov     word ptr [si], 0        ;Index := 0;
ForLoop:        cmp     word ptr [si], 10       ;For loop ends at ten.
                jg      ForDone
                call    dx                      ;Call thunk item.
                mov     word ptr [bx], 0        ;Store zero into item.
                inc     word ptr [si]           ;Index := Index + 1;
                jmp     ForLoop

ForDone:        pop     ax                      ;Restore AX.
                ret                             ;All Done!
PassByName      endp
```

You might call this routine with code that looks like the following:

```
                lea     si, I
                lea     dx, Thunk_A
                call    PassByName
                  .
                  .
                  .
Thunk_A         proc
                mov     bx, I
                shl     bx, 1
                lea     bx, A[bx]
                ret
Thunk_A         endp
```

The advantage to this scheme, over the one presented in the earlier section, is that you can call different thunks, not just the ItemThunk routine appearing in the earlier example.

11.5.8 Passing Parameters in Global Variables

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

```
                mov     ax, xxxx                ;Pass this parameter by value
                mov     Value1Proc1, ax
                mov     ax, offset yyyy         ;Pass this parameter by ref
                mov     word ptr Ref1Proc1, ax
                mov     ax, seg yyyy
                mov     word ptr Ref1Proc1+2, ax
                call    ThisProc
                  .
                  .
                  .
```

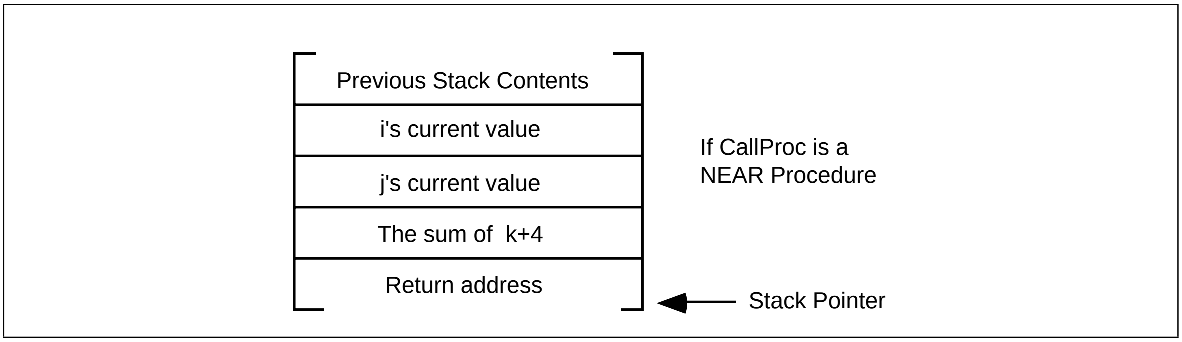

Figure 11.5 CallProc Stack Layout for a Near Procedure

```
ThisProc        proc    near
                push    es
                push    ax
                push    bx
                les     bx, Ref1Proc1           ;Get address of ref parm.
                mov     ax, Value1Proc1         ;Get value parameter
                mov     es:[bx], ax             ;Store into loc pointed at by
                pop     bx                      ; the ref parameter.
                pop     ax
                pop     es
                ret
ThisProc        endp
```

Passing parameters in global locations is inelegant and inefficient. Furthermore, if you use global variables in this fashion to pass parameters, the subroutines you write cannot use recursion (see “Recursion” on page 606). Fortunately, there are better parameter passing schemes for passing data in memory so you do not need to seriously consider this scheme.

11.5.9 Passing Parameters on the Stack

Most HLLs use the stack to pass parameters because this method is fairly efficient. To pass parameters on the stack, push them immediately before calling the subroutine. The subroutine then reads this data from the stack memory and operates on it appropriately. Consider the following Pascal procedure call:

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

Most Pascal compilers push their parameters onto the stack in the order that they appear in the parameter list. Therefore, the 80x86 code typically emitted for this subroutine call (assuming you’re passing the parameters by value) is

```
push    i
push    j
mov     ax, k
add     ax, 4
push    ax
call    CallProc
```

Upon entry into CallProc, the 80x86’s stack looks like that shown in Figure 11.5 (for a near procedure) or Figure 11.6 (for a far procedure).

You could gain access to the parameters passed on the stack by removing the data from the stack (Assuming a near procedure call):

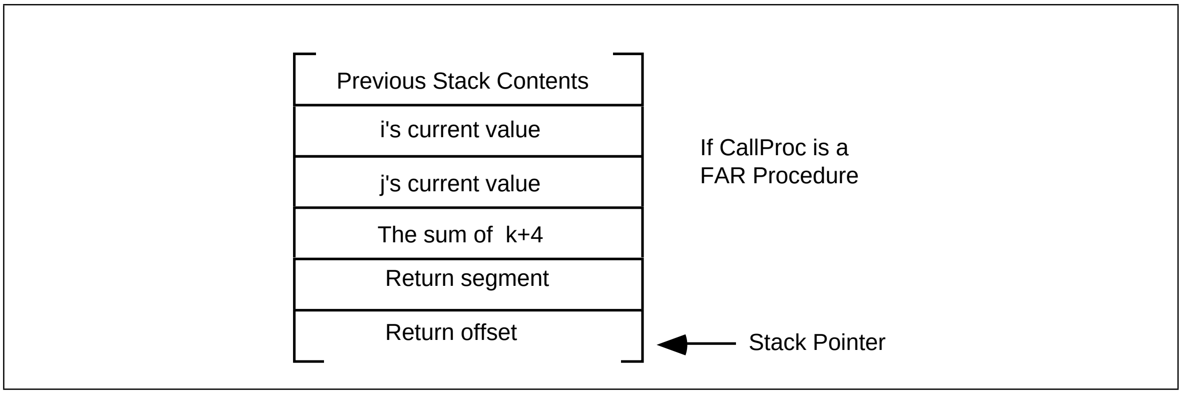

Figure 11.6 CallProc Stack Layout for a Far Procedure

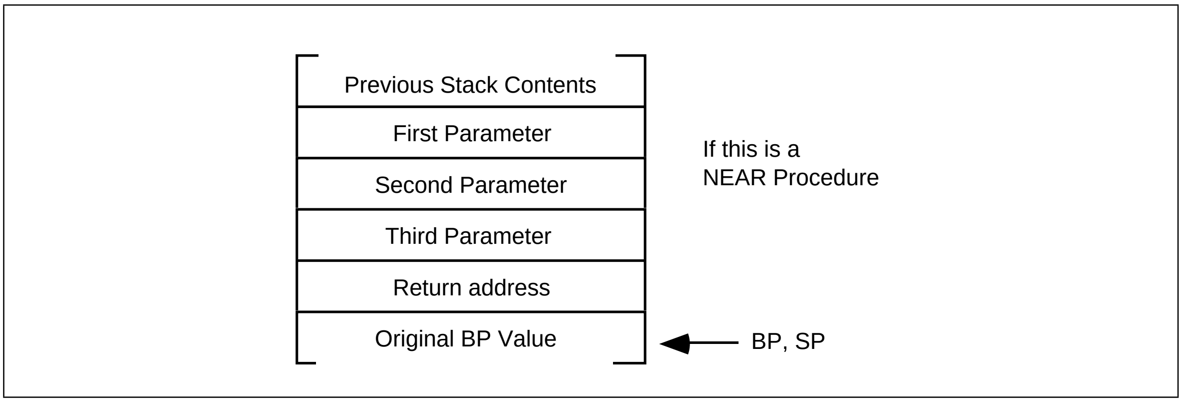

Figure 11.7 Accessing Parameters on the Stack

```
CallProc        proc    near
                pop     RtnAdrs
                pop     kParm
                pop     jParm
                pop     iParm
                push    RtnAdrs
                  .
                  .
                  .
                ret
CallProc        endp
```

There is, however, a better way. The 80x86's architecture allows you to use the bp (base pointer) register to access parameters passed on the stack. This is one of the reasons the disp[bp], [bp][di], [bp][si], disp[bp][si], and disp[bp][di] addressing modes use the stack segment rather than the data segment. The following code segment gives the *standard procedure entry and exit* code:

```
StdProc         proc    near
                push    bp
                mov     bp, sp
                  .
                  .
                  .
                pop     bp
                ret     ParmSize
StdProc         endp
```

ParmSize is the number of bytes of parameters pushed onto the stack before calling the procedure. In the CallProc procedure there were six bytes of parameters pushed onto the stack so ParmSize would be six.

Take a look at the stack immediately after the execution of mov bp, sp in StdProc. Assuming you've pushed three parameter words onto the stack, it should look something like shown in Figure 11.7.

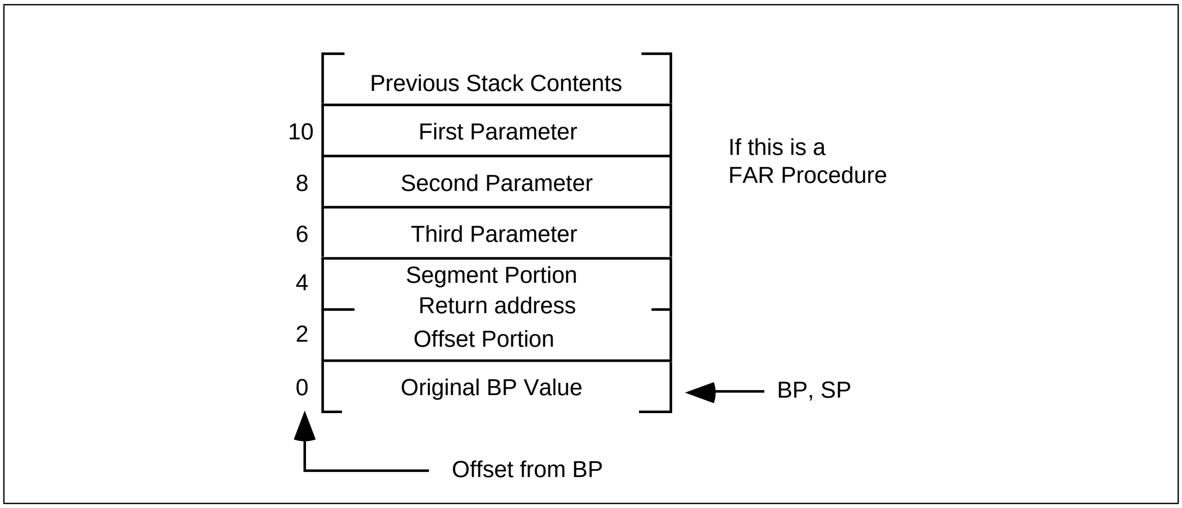

Figure 11.8 Accessing Parameters on the Stack in a Far Procedure

Now the parameters can be fetched by indexing off the bp register:

```
mov      ax, 8[bp]    ;Accesses the first parameter
mov      ax, 6[bp]    ;Accesses the second parameter
mov      ax, 4[bp]    ;Accesses the third parameter
```

When returning to the calling code, the procedure must remove these parameters from the stack. To accomplish this, pop the old bp value off the stack and execute a ret 6 instruction. This pops the return address off the stack and adds six to the stack pointer, effectively removing the parameters from the stack.

The displacements given above are for *near* procedures only. When calling a far procedure,

- 0[BP] will point at the old BP value,
- 2[BP] will point at the offset portion of the return address,
- 4[BP] will point at the segment portion of the return address, and
- 6[BP] will point at the last parameter pushed onto the stack.

The stack contents when calling a far procedure are shown in Figure 11.8.

This collection of parameters, return address, registers saved on the stack, and other items, is a *stack frame* or *activation record*.

When saving other registers onto the stack, always make sure that you save and set up bp before pushing the other registers. If you push the other registers before setting up bp, the offsets into the stack frame will change. For example, the following code disturbs the ordering presented above:

```
FunnyProc        proc     near
                 push     ax
                 push     bx
                 push     bp
                 mov      bp, sp
                   .
                   .
                   .
                 pop      bp
                 pop      bx
                 pop      ax
                 ret
FunnyProc        endp
```

Since this code pushes ax and bx before pushing bp and copying sp to bp, ax and bx appear in the activation record before the return address (that would normally start at location [bp+2]). As a result, the value of bx appears at location [bp+2] and the value of ax appears at location [bp+4]. This pushes the return address and other parameters farther up the stack as shown in Figure 11.9.

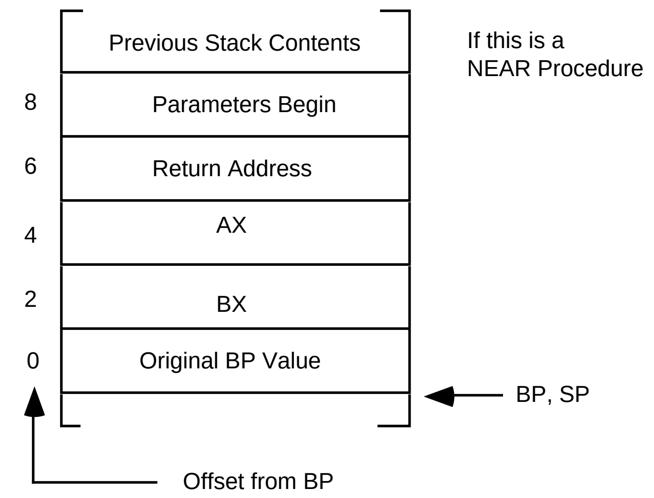

Figure 11.9 Messing up Offsets by Pushing Other Registers Before BP

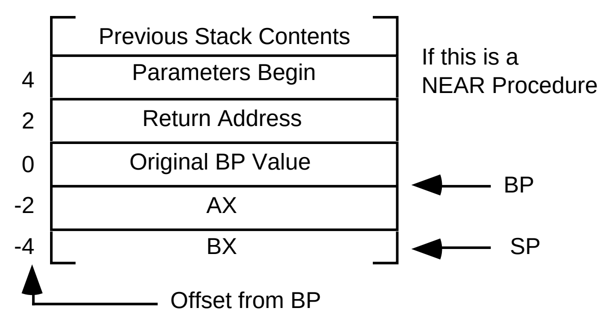

Figure 11.10 Keeping the Offsets Constant by Pushing BP First

Although this is a near procedure, the parameters don't begin until offset eight in the activation record. Had you pushed the ax and bx registers after setting up bp, the offset to the parameters would have been four (see Figure 11.10).

```
FunnyProc       proc    near
                push    bp
                mov     bp, sp
                push    ax
                push    bx
                  .
                  .
                  .
                pop     bx
                pop     ax
                pop     bp
                ret
FunnyProc       endp
```

Therefore, the push bp and mov bp, sp instructions should be the first two instructions any subroutine executes when it has parameters on the stack.

Accessing the parameters using expressions like [bp+6] can make your programs very hard to read and maintain. If you would like to use meaningful names, there are several ways to do so. One way to reference parameters by name is to use equates. Consider the following Pascal procedure and its equivalent 80x86 assembly language code:

```
        procedure xyz(var i:integer; j,k:integer);
        begin
                i := j+k;
        end;
```

Calling sequence:

```
                xyz(a,3,4);
```

Assembly language code:

```
xyz_i           equ     8[bp]           ;Use equates so we can reference
xyz_j           equ     6[bp]           ; symbolic names in the body of
xyz_k           equ     4[bp]           ; the procedure.
xyz             proc    near
                push    bp
                mov     bp, sp
                push    es
                push    ax
                push    bx
                les     bx, xyz_i       ;Get address of I into ES:BX
                mov     ax, xyz_j       ;Get J parameter
                add     ax, xyz_k       ;Add to K parameter
                mov     es:[bx], ax     ;Store result into I parameter
                pop     bx
                pop     ax
                pop     es
                pop     bp
                ret     8
xyz             endp
```

Calling sequence:

```
                mov     ax, seg a               ;This parameter is passed by
                push    ax                      ; reference, so pass its
                mov     ax, offset a            ; address on the stack.
                push    ax
                mov     ax, 3                   ;This is the second parameter
                push    ax
                mov     ax, 4                   ;This is the third parameter.
                push    ax
                call    xyz
```

On an 80186 or later processor you could use the following code in place of the above:

```
                push    seg a                   ;Pass address of “a” on the
                push    offset a                ; stack.
                push    3                       ;Pass second parm by val.
                push    4                       ;Pass third parm by val.
                call    xyz
```

Upon entry into the xyz procedure, before the execution of the les instruction, the stack looks like shown in Figure 11.11.

Since you’re passing I by reference, you must push its address onto the stack. This code passes reference parameters using 32 bit segmented addresses. Note that this code uses ret 8. Although there are three parameters on the stack, the reference parameter I consumes four bytes since it is a far address. Therefore there are eight bytes of parameters on the stack necessitating the ret 8 instruction.

Were you to pass I by reference using a near pointer rather than a far pointer, the code would look like the following:

```
xyz_i           equ     8[bp]           ;Use equates so we can reference
xyz_j           equ     6[bp]           ; symbolic names in the body of
xyz_k           equ     4[bp]           ; the procedure.
xyz             proc    near
                push    bp
                mov     bp, sp
                push    ax
                push    bx
                mov     bx, xyz_i       ;Get address of I into BX
```

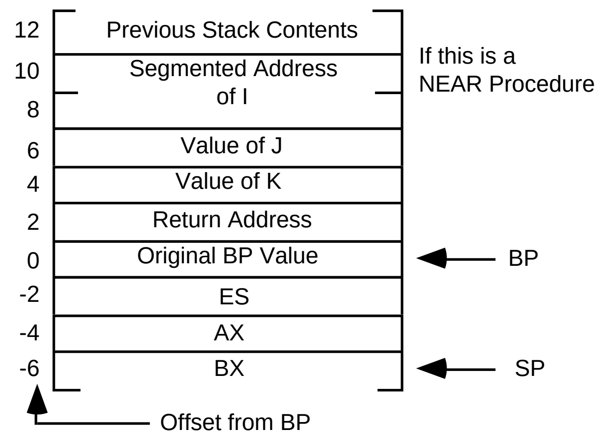

Figure 11.11 XYZ Stack Upon Procedure Entry

```
                mov     ax, xyz_j    ;Get J parameter
                add     ax, xyz_k    ;Add to K parameter
                mov     [bx], ax     ;Store result into I parameter
                pop     bx
                pop     ax
                pop     bp
                ret     6
xyz             endp
```

Note that since I's address on the stack is only two bytes (rather than four), this routine only pops six bytes when it returns.

Calling sequence:

```
                mov     ax, offset a ;Pass near address of a.
                push    ax
                mov     ax, 3        ;This is the second parameter
                push    ax
                mov     ax, 4        ;This is the third parameter.
                push    ax
                call    xyz
```

On an 80286 or later processor you could use the following code in place of the above:

```
                push    offset a               ;Pass near address of a.
                push    3                      ;Pass second parm by val.
                push    4                      ;Pass third parm by val.
                call    xyz
```

The stack frame for the above code appears in Figure 11.12.

When passing a parameter by value-returned or result, you pass an address to the procedure, exactly like passing the parameter by reference. The only difference is that you use a local copy of the variable within the procedure rather than accessing the variable indirectly through the pointer. The following implementations for xyz show how to pass I by value-returned and by result:

```
; xyz version using Pass by Value-Returned for xyz_i

xyz_i           equ     8[bp]        ;Use equates so we can reference
xyz_j           equ     6[bp]        ; symbolic names in the body of
xyz_k           equ     4[bp]        ; the procedure.

xyz             proc    near
                push    bp
                mov     bp, sp
                push    ax
                push    bx
```

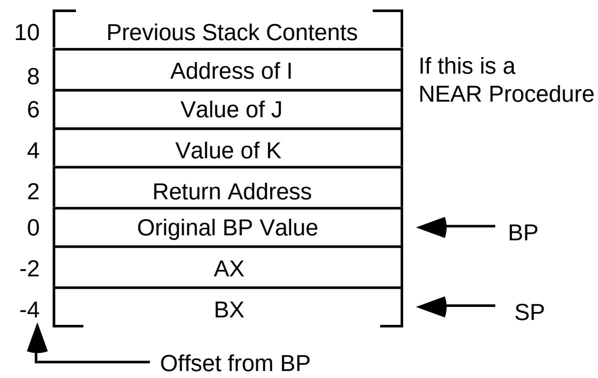

Figure 11.12 Passing Parameters by Reference Using Near Pointers Rather than Far Pointers

```
                push    cx              ;Keep local copy here.

                mov     bx, xyz_i       ;Get address of I into BX
                mov     cx, [bx]        ;Get local copy of I parameter.

                mov     ax, xyz_j       ;Get J parameter
                add     ax, xyz_k       ;Add to K parameter
                mov     cx, ax          ;Store result into local copy

                mov     bx, xyz_i       ;Get ptr to I, again
                mov     [bx], cx        ;Store result away.

                pop     cx
                pop     bx
                pop     ax
                pop     bp
                ret     6
xyz             endp
```

There are a couple of unnecessary mov instructions in this code. They are present only to precisely implement pass by value-returned parameters. It is easy to improve this code using pass by result parameters. The modified code is

```
; xyz version using Pass by Result for xyz_i

xyz_i           equ     8[bp]           ;Use equates so we can reference
xyz_j           equ     6[bp]           ; symbolic names in the body of
xyz_k           equ     4[bp]           ; the procedure.

xyz             proc    near
                push    bp
                mov     bp, sp
                push    ax
                push    bx
                push    cx              ;Keep local copy here.

                mov     ax, xyz_j       ;Get J parameter
                add     ax, xyz_k       ;Add to K parameter
                mov     cx, ax          ;Store result into local copy

                mov     bx, xyz_i       ;Get ptr to I, again
                mov     [bx], cx        ;Store result away.

                pop     cx
                pop     bx
                pop     ax
                pop     bp
                ret     6
xyz             endp
```

As with passing value-returned and result parameters in registers, you can improve the performance of this code using a modified form of pass by value. Consider the following implementation of xyz:

```
; xyz version using modified pass by value-result for xyz_i

xyz_i           equ     8[bp]       ;Use equates so we can reference
xyz_j           equ     6[bp]       ; symbolic names in the body of
xyz_k           equ     4[bp]       ; the procedure.

xyz             proc    near
                push    bp
                mov     bp, sp
                push    ax

                mov     ax, xyz_j   ;Get J parameter
                add     ax, xyz_k   ;Add to K parameter
                mov     xyz_i, ax   ;Store result into local copy

                pop     ax
                pop     bp
                ret     4           ;Note that we do not pop I parm.
xyz             endp
```

The calling sequence for this code is

```
                push    a           ;Pass a's value to xyz.
                push    3           ;Pass second parameter by val.
                push    4           ;Pass third parameter by val.
                call    xyz
                pop     a
```

Note that a pass by result version wouldn't be practical since you have to push *something* on the stack to make room for the local copy of I inside xyz. You may as well push the value of a on entry even though the xyz procedure ignores it. This procedure pops only *four* bytes off the stack on exit. This leaves the value of the I parameter on the stack so that the calling code can store it away to the proper destination.

To pass a parameter by name on the stack, you simply push the address of the thunk. Consider the following pseudo-Pascal code:

```
procedure swap(name Item1, Item2:integer);
var temp:integer;
begin

       temp := Item1;
       Item1 := Item2;
       Item2 := Temp;

end;
```

If swap is a near procedure, the 80x86 code for this procedure could look like the following (note that this code has been slightly optimized and does not following the exact sequence given above):

```
; swap-         swaps two parameters passed by name on the stack.
;               Item1 is passed at address [bp+6], Item2 is passed
;               at address [bp+4]

wp              textequ  <word ptr>
swap_Item1      equ     [bp+6]
swap_Item2      equ     [bp+4]

swap            proc    near
                push    bp
                mov     bp, sp
                push    ax                      ;Preserve temp value.
                push    bx                      ;Preserve bx.
                call    wp swap_Item1           ;Get adrs of Item1.
                mov     ax, [bx]                ;Save in temp (AX).
                call    wp swap_Item2           ;Get adrs of Item2.
                xchg    ax, [bx]                ;Swap temp <-> Item2.
                call    wp swap_Item1           ;Get adrs of Item1.
```

```
                mov      [bx], ax                ;Save temp in Item1.
                pop      bx                      ;Restore bx.
                pop      ax                      ;Restore ax.
                ret      4                       ;Return and pop Item1/2.
swap            endp
```

Some sample calls to swap follow:

```
; swap(A[i], i) -- 8086 version.

                lea      ax, thunk1
                push     ax
                lea      ax, thunk2
                push     ax
                call     swap

; swap(A[i],i) -- 80186 & later version.

                push     offset thunk1
                push     offset thunk2
                call     swap

                 .
                 .
                 .

; Note: this code assumes A is an array of two byte integers.

thunk1          proc     near
                mov      bx, i
                shl      bx, 1
                lea      bx, A[bx]
                ret
thunk1          endp

thunk2          proc     near
                lea      bx, i
                ret
thunk2          endp
```

The code above assumes that the thunks are near procs that reside in the same segment as the swap routine. If the thunks are far procedures the caller must pass far addresses on the stack and the swap routine must manipulate far addresses. The following implementation of swap, thunk1, and thunk2 demonstrate this.

```
; swap-         swaps two parameters passed by name on the stack.
;               Item1 is passed at address [bp+10], Item2 is passed
;               at address [bp+6]

swap_Item1      equ      [bp+10]
swap_Item2      equ      [bp+6]
dp              textequ  <dword ptr>

swap            proc     far
                push     bp
                mov      bp, sp
                push     ax                      ;Preserve temp value.
                push     bx                      ;Preserve bx.
                push     es                      ;Preserve es.
                call     dp swap_Item1           ;Get adrs of Item1.
                mov      ax, es:[bx]             ;Save in temp (AX).
                call     dp swap_Item2           ;Get adrs of Item2.
                xchg     ax, es:[bx]             ;Swap temp <-> Item2.
                call     dp swap_Item1           ;Get adrs of Item1.
                mov      es:[bx], ax             ;Save temp in Item1.
                pop      es                      ;Restore es.
                pop      bx                      ;Restore bx.
                pop      ax                      ;Restore ax.
                ret      8                       ;Return and pop Item1, Item2.
swap            endp
```

Some sample calls to swap follow:

```
; swap(A[i], i) -- 8086 version.

                mov     ax, seg thunk1
                push    ax
                lea     ax, thunk1
                push    ax
                mov     ax, seg thunk2
                push    ax
                lea     ax, thunk2
                push    ax
                call    swap

; swap(A[i],i) -- 80186 & later version.

                push    seg thunk1
                push    offset thunk1
                push    seg thunk2
                push    offset thunk2
                call    swap
                  .
                  .
                  .

; Note:         this code assumes A is an array of two byte integers.
;               Also note that we do not know which segment(s) contain
;               A and I.

thunk1          proc    far
                mov     bx, seg A     ;Need to return seg A in ES.
                push    bx            ;Save for later.
                mov     bx, seg i     ;Need segment of I in order
                mov     es, bx        ; to access it.
                mov     bx, es:i      ;Get I's value.
                shl     bx, 1
                lea     bx, A[bx]
                pop     es            ;Return segment of A[I] in es.
                ret
thunk1          endp

thunk2          proc    near
                mov     bx, seg i     ;Need to return I's seg in es.
                mov     es, bx
                lea     bx, i
                ret
thunk2          endp
```

Passing parameters by lazy evaluation is left for the programming projects.

Additional information on activation records and stack frames appears later in this chapter in the section on local variables.

11.5.10 Passing Parameters in the Code Stream

Another place where you can pass parameters is in the code stream immediately after the call instruction. The print routine in the UCR Standard Library package provides an excellent example:

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

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

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

```
MyPrint         proc    near
                push    bp
                mov     bp, sp
                push    bx
                push    ax
                mov     bx, 2[bp]      ;Load return address into BX
PrintLp:        mov     al, cs:[bx]    ;Get next character
                cmp     al, 0          ;Check for end of string
                jz      EndStr
                putc                   ;If not end, print this char
                inc     bx             ;Move on to the next character
                jmp     PrintLp

EndStr:         inc     bx             ;Point at first byte beyond zero
                mov     2[bp], bx      ;Save as new return address
                pop     ax
                pop     bx
                pop     bp
                ret
MyPrint         endp
```

This procedure begins by pushing all the affected registers onto the stack. It then fetches the return address, at offset 2[BP], and prints each successive character until encountering a zero byte. Note the presence of the cs: segment override prefix in the mov al, cs:[bx] instruction. Since the data is coming from the code segment, this prefix guarantees that MyPrint fetches the character data from the proper segment. Upon encountering the zero byte, MyPrint points bx at the first byte beyond the zero. This is the address of the first instruction following the zero terminating byte. The CPU uses this value as the new return address. Now the execution of the ret instruction returns control to the instruction following the string.

The above code works great if MyPrint is a near procedure. If you need to call MyPrint from a different segment you will need to create a far procedure. Of course, the major difference is that a far return address will be on the stack at that point – you will need to use a far pointer rather than a near pointer. The following implementation of MyPrint handles this case.

```
MyPrint         proc    far
                push    bp
                mov     bp, sp
                push    bx             ;Preserve ES, AX, and BX
                push    ax
                push    es

                les     bx, 2[bp]      ;Load return address into ES:BX
PrintLp:        mov     al, es:[bx]    ;Get next character
                cmp     al, 0          ;Check for end of string
                jz      EndStr
                putc                   ;If not end, print this char
                inc     bx             ;Move on to the next character
                jmp     PrintLp

EndStr:         inc     bx             ;Point at first byte beyond zero
                mov     2[bp], bx      ;Save as new return address
                pop     es
                pop     ax
                pop     bx
                pop     bp
                ret
MyPrint         endp
```

Note that this code does not store es back into location [bp+4]. The reason is quite simple – es does not change during the execution of this procedure; storing es into location [bp+4] would not change the value at that location. You will notice that this version of MyPrint fetches each character from location es:[bx] rather than cs:[bx]. This is because the string you're printing is in the caller's segment, that might not be the same segment containing MyPrint.

Besides showing how to pass parameters in the code stream, the MyPrint routine also exhibits another concept: *variable length parameters*. The string following the call can be any practical length. The zero terminating byte marks the end of the parameter list. There are two easy ways to handle variable length parameters. Either use some special terminating value (like zero) or you can pass a special length value that tells the subroutine how many parameters you are passing. Both methods have their advantages and disadvantages. Using a special value to terminate a parameter list requires that you choose a value that never appears in the list. For example, MyPrint uses zero as the terminating value, so it cannot print the NULL character (whose ASCII code is zero). Sometimes this isn't a limitation. Specifying a special length parameter is another mechanism you can use to pass a variable length parameter list. While this doesn't require any special codes or limit the range of possible values that can be passed to a subroutine, setting up the length parameter and maintaining the resulting code can be a real nightmare[5].

Although passing parameters in the code stream is an ideal way to pass variable length parameter lists, you can pass fixed length parameter lists as well. The code stream is an excellent place to pass constants (like the string constants passed to MyPrint) and reference parameters. Consider the following code that expects three parameters by reference:

Calling sequence:

```
                call    AddEm
                word    I,J,K
```

Procedure:

```
AddEm           proc    near
                push    bp
                mov     bp, sp
                push    si
                push    bx
                push    ax
                mov     si, [bp+2]              ;Get return address
                mov     bx, cs:[si+2]           ;Get address of J
                mov     ax, [bx]                ;Get J's value
                mov     bx, cs:[si+4]           ;Get address of K
                add     ax, [bx]                ;Add in K's value
                mov     bx, cs:[si]             ;Get address of I
                mov     [bx], ax                ;Store result
                add     si, 6                   ;Skip past parms
                mov     [bp+2], si              ;Save return address
                pop     ax
                pop     bx
                pop     si
                pop     bp
                ret
AddEm           endp
```

This subroutine adds J and K together and stores the result into I. Note that this code uses 16 bit near pointers to pass the addresses of I, J, and K to AddEm. Therefore, I, J, and K must be in the current data segment. In the example above, AddEm is a near procedure. Had it been a far procedure it would have needed to fetch a four byte pointer from the stack rather than a two byte pointer. The following is a far version of AddEm:

```
AddEm           proc    far
                push    bp
                mov     bp, sp
                push    si
                push    bx
                push    ax
                push    es
                les     si, [bp+2]              ;Get far ret adrs into es:si
                mov     bx, es:[si+2]           ;Get address of J
                mov     ax, [bx]                ;Get J's value
```

5. Especially if the parameter list changes frequently.

```
                mov     bx, es:[si+4]           ;Get address of K
                add     ax, [bx]                ;Add in K's value
                mov     bx, es:[si]             ;Get address of I
                mov     [bx], ax                ;Store result
                add     si, 6                   ;Skip past parms
                mov     [bp+2], si              ;Save return address
                pop     es
                pop     ax
                pop     bx
                pop     si
                pop     bp
                ret
AddEm           endp
```

In both versions of AddEm, the pointers to I, J, and K passed in the code stream are near pointers. Both versions assume that I, J, and K are all in the current data segment. It is possible to pass far pointers to these variables, or even near pointers to some and far pointers to others, in the code stream. The following example isn't quite so ambitious, it is a near procedure that expects far pointers, but it does show some of the major differences. For additional examples, see the exercises.

Callling sequence:

```
                call    AddEm
                dword   I,J,K
```

Code:

```
AddEm           proc    near
                push    bp
                mov     bp, sp
                push    si
                push    bx
                push    ax
                push    es
                mov     si, [bp+2]              ;Get near ret adrs into si
                les     bx, cs:[si+2]           ;Get address of J into es:bx
                mov     ax, es:[bx]             ;Get J's value
                les     bx, cs:[si+4]           ;Get address of K
                add     ax, es:[bx]             ;Add in K's value
                les     bx, cs:[si]             ;Get address of I
                mov     es:[bx], ax             ;Store result
                add     si, 12                  ;Skip past parms
                mov     [bp+2], si              ;Save return address
                pop     es
                pop     ax
                pop     bx
                pop     si
                pop     bp
                ret
AddEm           endp
```

Note that there are 12 bytes of parameters in the code stream this time around. This is why this code contains an add si, 12 instruction rather than the add si, 6 appearing in the other versions.

In the examples given to this point, MyPrint expects a pass by value parameter, it prints the actual characters following the call, and AddEm expects three pass by reference parameters – their addresses follow in the code stream. Of course, you can also pass parameters by value-returned, by result, by name, or by lazy evaluation in the code stream as well. The next example is a modification of AddEm that uses pass by result for I, pass by value-returned for J, and pass by name for K. This version is slightly differerent insofar as it modifies J as well as I, in order to justify the use of the value-returned parameter.

```
; AddEm(Result I:integer; ValueResult J:integer; Name K);
;
;       Computes           I:= J;
;                          J := J+K;
;
; Presumes all pointers in the code stream are near pointers.

AddEm           proc     near
                push     bp
                mov      bp, sp
                push     si                     ;Pointer to parameter block.
                push     bx                     ;General pointer.
                push     cx                     ;Temp value for I.
                push     ax                     ;Temp value for J.

                mov      si, [bp+2]             ;Get near ret adrs into si

                mov      bx, cs:[si+2]          ;Get address of J into bx
                mov      ax, es:[bx]            ;Create local copy of J.
                mov      cx, ax                 ;Do I:=J;

                call     word ptr cs:[si+4]     ;Call thunk to get K's adrs
                add      ax, [bx]               ;Compute J := J + K

                mov      bx, cs:[si]            ;Get address of I and store
                mov      [bx], cx               ; I away.

                mov      bx, cs:[si+2]          ;Get J's address and store
                mov      [bx], ax               ; J's value away.

                add      si, 6                  ;Skip past parms
                mov      [bp+2], si             ;Save return address
                pop      ax
                pop      cx
                pop      bx
                pop      si
                pop      bp
                ret
AddEm           endp
```

Example calling sequences:

```
; AddEm(I,J,K)

                call     AddEm
                word     I,J,KThunk

; AddEm(I,J,A[I])

                call     AddEm
                word     I,J,AThunk
                 .
                 .
                 .
KThunk          proc     near
                lea      bx, K
                ret
KThunk          endp

AThunk          proc     near
                mov      bx, I
                shl      bx, 1
                lea      bx, A[bx]
                ret
AThunk          endp
```

Note: had you passed I by reference, rather than by result, in this example, the call

AddEm(I,J,A[i])

would have produced different results. Can you explain why?

Passing parameters in the code stream lets you perform some really clever tasks. The following example is considerably more complex than the others in this section, but it

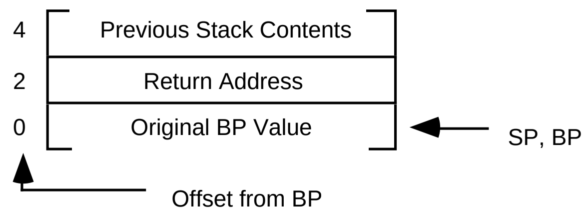

Figure 11.13 Stack Upon Entry into the ForStmt Procedure

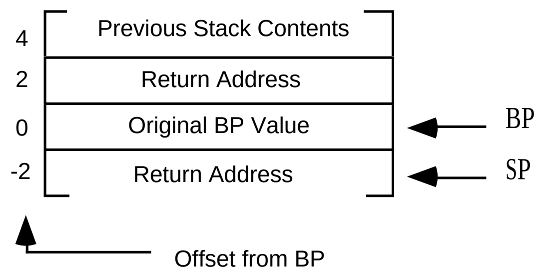

Figure 11.14 Stack Just Before Leaving the ForStmt Procedure

demonstrates the power of passing parameters in the code stream and, despite the complexity of this example, how they can simplify your programming tasks.

The following two routines implement a for/next statement, similar to that in BASIC, in assembly language. The calling sequence for these routines is the following:

```
call     ForStmt
word     «LoopControlVar», «StartValue», «EndValue»
  .
  .
« loop body statements»
  .
  .
call     Next
```

This code sets the loop control variable (whose near address you pass as the first parameter, by reference) to the starting value (passed by value as the second parameter). It then begins execution of the loop body. Upon executing the call to Next, this program would increment the loop control variable and then compare it to the ending value. If it is less than or equal to the ending value, control would return to the beginning of the loop body (the first statement following the word directive). Otherwise it would continue execution with the first statement past the call to Next.

Now you're probably wondering, "How on earth does control transfer to the beginning of the loop body?" After all, there is no label at that statement and there is no control transfer instruction instruction that jumps to the first statement after the word directive. Well, it turns out you can do this with a little tricky stack manipulation. Consider what the stack will look like upon entry into the ForStmt routine, after pushing bp onto the stack (see Figure 11.13).

Normally, the ForStmt routine would pop bp and return with a ret instruction, which removes ForStmt's activation record from the stack. Suppose, instead, ForStmt executes the following instructions:

```
add      word ptr 2[b], 2        ;Skip the parameters.
push     [bp+2]                  ;Make a copy of the rtn adrs.
mov      bp, [bp]                ;Restore bp's value.
ret                              ;Return to caller.
```

Just before the ret instruction above, the stack has the entries shown in Figure 11.14.

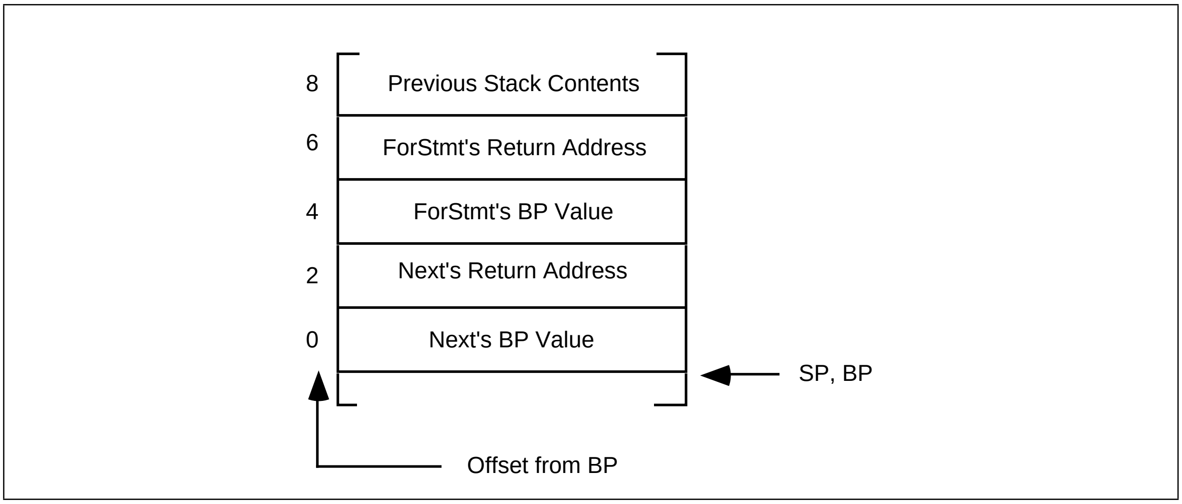

Figure 11.15 The Stack upon Entering the Next Procedure

Upon executing the ret instruction, ForStmt will return to the proper return address *but it will leave its activation record on the stack!*

After executing the statements in the loop body, the program calls the Next routine. Upon initial entry into Next (and setting up bp), the stack contains the entries appearing in Figure 11.15[6].

The important thing to see here is that ForStmt's return address, that points at the first statement past the word directive, is still on the stack and available to Next at offset [bp+6]. Next can use this return address to gain access to the parameters and return to the appropriate spot, if necessary. Next increments the loop control variable and compares it to the ending value. If the loop control variable's value is less than the ending value, Next pops its return address off the stack and returns through ForStmt's return address. If the loop control variable is greater than the ending value, Next returns through its own return address and removes ForStmt's activation record from the stack. The following is the code for Next and ForStmt:

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

dseg              segment   para public 'data'
I                 word      ?
J                 word      ?
dseg              ends

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

wp                textequ   <word ptr>

ForStmt           proc      near
                  push      bp
                  mov       bp, sp
                  push      ax
                  push      bx
                  mov       bx, [bp+2]   ;Get return address
                  mov       ax, cs:[bx+2];Get starting value
                  mov       bx, cs:[bx]  ;Get address of var
                  mov       [bx], ax     ;var := starting value
                  add       wp [bp+2], 6 ;Skip over parameters
                  pop       bx
```

6. Assuming the loop does not push anything onto the stack, or pop anything off the stack. Should either case occur, the ForStmt / Next loop would not work properly.

```
                pop     ax
                push    [bp+2]      ;Copy return address
                mov     bp, [bp]    ;Restore bp
                ret                 ;Leave Act Rec on stack
ForStmt         endp

Next            proc near
                push    bp
                mov     bp, sp
                push    ax
                push    bx
                mov     bx, [bp+6]  ;ForStmt's rtn adrs
                mov     ax, cs:[bx-2];Ending value
                mov     bx, cs:[bx-6];Ptr to loop ctrl var
                inc     wp [bx]     ;Bump up loop ctrl
                cmp     ax, [bx]    ;Is end val < loop ctrl?
                jl      QuitLoop

; If we get here, the loop control variable is less than or equal
; to the ending value. So we need to repeat the loop one more time.
; Copy ForStmt's return address over our own and then return,
; leaving ForStmt's activation record intact.

                mov     ax, [bp+6]  ;ForStmt's return address
                mov     [bp+2], ax  ;Overwrite our return address
                pop     bx
                pop     ax
                pop     bp          ;Return to start of loop body
                ret

; If we get here, the loop control variable is greater than the
; ending value, so we need to quit the loop (by returning to Next's
; return address) and remove ForStmt's activation record.

QuitLoop:       pop     bx
                pop     ax
                pop     bp
                ret     4
Next            endp

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

                call    ForStmt
                word    I,1,5
                call    ForStmt
                word    J,2,4
                printf
                byte    "I=%d, J=%d\n",0
                dword   I,J

                call    Next        ;End of J loop
                call    Next        ;End of I loop
                print
                byte    "All Done!",cr,lf,0

Quit:           ExitPgm
Main            endp
cseg            ends
sseg            segment para stack 'stack'
stk             byte    1024 dup ("stack ")
sseg            ends
zzzzzzseg       segment para public 'zzzzzz'
LastBytes       byte    16 dup (?)
zzzzzzseg       ends
                end     Main
```

The example code in the main program shows that these for loops nest exactly as you would expect in a high level language like BASIC, Pascal, or C. Of course, this is not a particularly good way to construct a for loop in assembly language. It is many times slower than using the standard loop generation techniques (see "Loops" on page 531 for more details on that). Of course, if you don't care about speed, this is a perfectly good way to implement a loop. It is certainly easier to read and understand than the traditional methods for creating a for loop. For another (more efficient) implementation of the for loop, check out the ForLp macros in Chapter Eight (see "A Sample Macro to Implement For Loops" on page 409).

The code stream is a very convenient place to pass parameters. The UCR Standard Library makes considerable use of this parameter passing mechanism to make it easy to call certain routines. Printf is, perhaps, the most complex example, but other examples (especially in the string library) abound.

Despite the convenience, there are some disadvantages to passing parameters in the code stream. First, if you fail to provide the exact number of parameters the procedure requires, the subroutine will get very confused. Consider the UCR Standard Library print routine. It prints a string of characters up to a zero terminating byte and then returns control to the first instruction following the zero terminating byte. If you leave off the zero terminating byte, the print routine happily prints the following opcode bytes as ASCII characters until it finds a zero byte. Since zero bytes often appear in the middle of an instruction, the print routine might return control into the middle of some other instruction. This will probably crash the machine. Inserting an extra zero, which occurs more often than you might think, is another problem programmers have with the print routine. In such a case, the print routine would return upon encountering the first zero byte and attempt to execute the following ASCII characters as machine code. Once again, this usually crashes the machine.

Another problem with passing parameters in the code stream is that it takes a little longer to access such parameters. Passing parameters in the registers, in global variables, or on the stack is slightly more efficient, especially in short routines. Nevertheless, accessing parameters in the code stream isn't extremely slow, so the convenience of such parameters may outweigh the cost. Furthermore, many routines (print is a good example) are so slow anyway that a few extra microseconds won't make any difference.

11.5.11 Passing Parameters via a Parameter Block

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

Calling sequence:

```
ParmBlock       dword   I
I               word    ?                       ;I, J, and K must appear in
J               word    ?                       ; this order.
K               word    ?
                  .
                  .
                  .
                les     bx, ParmBlock
                call    AddEm
                  .
                  .
                  .
AddEm           proc    near
                push    ax
                mov     ax, es:2[bx]            ;Get J's value
                add     ax, es:4[bx]            ;Add in K's value
                mov     es:[bx], ax             ;Store result in I
                pop     ax
                ret
AddEm           endp
```

Note that you must allocate the three parameters in contiguous memory locations.

This form of parameter passing works well when passing several parameters by reference, because you can initialize pointers to the parameters directly within the assembler. For example, suppose you wanted to create a subroutine rotate to which you pass four parameters by reference. This routine would copy the second parameter to the first, the third to the second, the fourth to the third, and the first to the fourth. Any easy way to accomplish this in assembly is

```
; Rotate-         On entry, BX points at a parameter block in the data
;                 segment that points at four far pointers. This code
;                 rotates the data referenced by these pointers.

Rotate            proc      near
                  push      es                          ;Need to preserve these
                  push      si                          ; registers
                  push      ax

                  les       si, [bx+4]                  ;Get ptr to 2nd var
                  mov       ax, es:[si]                 ;Get its value
                  les       si, [bx]                    ;Get ptr to 1st var
                  xchg      ax, es:[si]                 ;2nd->1st, 1st->ax
                  les       si, [bx+12]                 ;Get ptr to 4th var
                  xchg      ax, es:[si]                 ;1st->4th, 4th->ax
                  les       si, [bx+8]                  ;Get ptr to 3rd var
                  xchg      ax, es:[si]                 ;4th->3rd, 3rd->ax
                  les       si, [bx+4]                  ;Get ptr to 2nd var
                  mov       es:[si], ax                 ;3rd -> 2nd

                  pop       ax
                  pop       si
                  pop       es
                  ret
Rotate            endp
```

To call this routine, you pass it a pointer to a group of four far pointers in the bx register. For example, suppose you wanted to rotate the first elements of four different arrays, the second elements of those four arrays, and the third elements of those four arrays. You could do this with the following code:

```
                  lea       bx, RotateGrp1
                  call      Rotate
                  lea       bx, RotateGrp2
                  call      Rotate
                  lea       bx, RotateGrp3
                  call      Rotate
                    .
                    .
                    .
RotateGrp1        dword     ary1[0], ary2[0], ary3[0], ary4[0]
RotateGrp2        dword     ary1[2], ary2[2], ary3[2], ary4[2]
RotateGrp3        dword     ary1[4], ary2[4], ary3[4], ary4[4]
```

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

Although beginning assembly language programmers rarely use parameter blocks, they certainly have their place. Some of the IBM PC BIOS and MS-DOS functions use this parameter passing mechanism. Parameter blocks, since you can initialize their values during assembly (using byte, word, etc.), provide a fast, efficient way to pass parameters to a procedure.

Of course, you can pass parameters by value, reference, value-returned, result, or by name in a parameter block. The following piece of code is a modification of the Rotate procedure above where the first parameter is passed by value (its value appears inside the parameter block), the second is passed by reference, the third by value-returned, and the fourth by name (there is no pass by result since Rotate needs to read and write all values). For simplicity, this code uses near pointers and assumes all variables appear in the data segment:

```
; Rotate-        On entry, DI points at a parameter block in the data
;                segment that points at four pointers. The first is
;                a value parameter, the second is passed by reference,
;                the third is passed by value/return, the fourth is
;                passed by name.

Rotate           proc     near
                 push     si                       ;Used to access ref parms
                 push     ax                       ;Temporary
                 push     bx                       ;Used by pass by name parm
                 push     cx                       ;Local copy of val/ret parm

                 mov      si, [di+4]               ;Get a copy of val/ret parm
                 mov      cx, [si]

                 mov      ax, [di]                 ;Get 1st (value) parm
                 call     word ptr [di+6]          ;Get ptr to 4th var
                 xchg     ax, [bx]                 ;1st->4th, 4th->ax
                 xchg     ax, cx                   ;4th->3rd, 3rd->ax
                 mov      bx, [di+2]               ;Get adrs of 2nd (ref) parm
                 xchg     ax, [bx]                 ;3rd->2nd, 2nd->ax
                 mov      [di], ax                 ;2nd->1st

                 mov      bx, [di+4]               ;Get ptr to val/ret parm
                 mov      [bx], cx                 ;Save val/ret parm away.

                 pop      cx
                 pop      bx
                 pop      ax
                 pop      si
                 ret
Rotate           endp
```

A reasonable example of a call to this routine might be:

```
I                word     10
J                word     15
K                word     20
RotateBlk        word     25, I, J, KThunk
                   .
                   .
                   .
                 lea      di, RotateBlk
                 call     Rotate
                   .
                   .
                   .
KThunk           proc     near
                 lea      bx, K
                 ret
KThunk           endp
```

11.6 Function Results

Functions return a result, which is nothing more than a result parameter. In assembly language, there are very few differences between a procedure and a function. That is probably why there aren't any "func" or "endf" directives. Functions and procedures are usually different in HLLs, function calls appear only in expressions, subroutine calls as statements[7]. Assembly language doesn't distinguish between them.

You can return function results in the same places you pass and return parameters. Typically, however, a function returns only a single value (or single data structure) as the

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

function result. The methods and locations used to return function results is the subject of the next three sections.

11.6.1 Returning Function Results in a Register

Like parameters, the 80x86's registers are the best place to return function results. The getc routine in the UCR Standard Library is a good example of a function that returns a value in one of the CPU's registers. It reads a character from the keyboard and returns the ASCII code for that character in the al register. Generally, functions return their results in the following registers:

Use	First Last	
Bytes:	al, ah, dl, dh, cl, ch, bl, bh	
Words:	ax, dx, cx, si, di, bx	
Double words:	dx:ax	On pre-80386
	eax, edx, ecx, esi, edi, ebx	On 80386 and later.
16-bitOffsets:	bx, si, di, dx	
32-bit Offsets	ebx, esi , edi, eax, ecx, edx	
Segmented Pointers:	es:di, es:bx, dx:ax, es:si	Do not use DS.

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

11.6.2 Returning Function Results on the Stack

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

```
function PasFunc(i,j,k:integer):integer;
begin
        PasFunc := i+j+k;
end;

m := PasFunc(2,n,l);
```

In assembly:

```
PasFunc_rtn     equu    10[bp]
PasFunc_i       equ     8[bp]
PasFunc_j       equ     6[bp]
PasFunc_k       equ     4[bp]

PasFunc         proc    near
                push    bp
                mov     bp, sp
                push    ax
                mov     ax, PasFunc_i
                add     ax, PasFunc_j
                add     ax, PasFunc_k
                mov     PasFunc_rtn, ax
                pop     ax
                pop     bp
                ret     6
PasFunc         endp
```

Calling sequence:

```
push    ax          ;Space for function return result
mov     ax, 2
push    ax
push    n
push    l
call    PasFunc
pop     ax          ;Get function return result
```

On an 80286 or later processor you could also use the code:

```
push    ax          ;Space for function return result
push    2
push    n
push    l
call    PasFunc
pop     ax          ;Get function return result
```

Although the caller pushed eight bytes of data onto the stack, PasFunc only removes six. The first "parameter" on the stack is the function result. The function must leave this value on the stack when it returns.

11.6.3 Returning Function Results in Memory Locations

Another reasonable place to return function results is in a known memory location. You can return function values in global variables or you can return a pointer (presumably in a register or a register pair) to a parameter block. This process is virtually identical to passing parameters to a procedure or function in global variables or via a parameter block.

Returning parameters via a pointer to a parameter block is an excellent way to return large data structures as function results. If a function returns an entire array, the best way to return this array is to allocate some storage, store the data into this area, and leave it up to the calling routine to deallocate the storage. Most high level languages that allow you to return large data structures as function results use this technique.

Of course, there is very little difference between returning a function result in memory and the pass by result parameter passing mechanism. See "Pass by Result" on page 576 for more details.

11.7 Side Effects

A *side effect* is any computation or operation by a procedure that isn't the primary purpose of that procedure. For example, if you elect not to preserve all affected registers within a procedure, the modification of those registers is a side effect of that procedure. Side effect programming, that is, the practice of using a procedure's side effects, is very dangerous. All too often a programmer will rely on a side effect of a procedure. Later modifications may change the side effect, invalidating all code relying on that side effect. This can make your programs hard to debug and maintain. Therefore, you should avoid side effect programming.

Perhaps some examples of side effect programming will help enlighten you to the difficulties you may encounter. The following procedure zeros out an array. For efficiency reasons, it makes the caller responsible for preserving necessary registers. As a result, one side effect of this procedure is that the bx and cx registers are modified. In particular, the cx register contains zero upon return.

```
ClrArray        proc    near
                lea     bx, array
                mov     cx, 32
ClrLoop:        mov     word ptr [bx], 0
                inc     bx
                inc     bx
                loop    ClrLoop
                ret
ClrArray        endp
```

If your code expects cx to contain zero after the execution of this subroutine, you would be relying on a side effect of the ClrArray procedure. The main purpose behind this code is zeroing out an array, not setting the cx register to zero. Later, if you modify the ClrArray procedure to the following, your code that depends upon cx containing zero would no longer work properly:

```
ClrArray        proc    near
                lea     bx, array
ClrLoop:        mov     word ptr [bx], 0
                inc     bx
                inc     bx
                cmp     bx, offset array+32
                jne     ClrLoop
                ret
ClrArray        endp
```

So how can you avoid the pitfalls of side effect programming in your procedures? By carefully structuring your code and paying close attention to exactly how your calling code and the subservient procedures interface with one another. These rules can help you avoid problems with side effect programming:

- Always properly document the input and output conditions of a procedure. Never rely on any other entry or exit conditions other than these documented operations.
- Partition your procedures so that they compute a single value or execute a single operation. Subroutines that do two or more tasks are, by definition, producing side effects unless every invocation of that subroutine requires all the computations and operations.
- When updating the code in a procedure, make sure that it still obeys the entry and exit conditions. If not, either modify the program so that it does or update the documentation for that procedure to reflect the new entry and exit conditions.
- Avoid passing information between routines in the CPU's flag register. Passing an error status in the carry flag is about as far as you should ever go. Too many instructions affect the flags and it's too easy to foul up a return sequence so that an important flag is modified on return.
- Always save and restore all registers a procedure modifies.
- Avoid passing parameters and function results in global variables.
- Avoid passing parameters by reference (with the intent of modifying them for use by the calling code).

These rules, like all other rules, were meant to be broken. Good programming practices are often sacrificed on the altar of efficiency. There is nothing wrong with breaking these rules as often as you feel necessary. However, your code will be difficult to debug and maintain if you violate these rules often. But such is the price of efficiency[8]. Until you gain enough experience to make a judicious choice about the use of side effects in your programs, you should avoid them. More often than not, the use of a side effect will cause more problems than it solves.

8. This is not just a snide remark. Expert programmers who have to wring the last bit of performance out of a section of code often resort to poor programming practices in order to achieve their goals. They are prepared, however, to deal with the problems that are often encountered in such situations and they are a lot more careful when dealing with such code.

11.8 Local Variable Storage

Sometimes a procedure will require temporary storage, that it no longer requires when the procedure returns. You can easily allocate such local variable storage on the stack.

The 80x86 supports local variable storage with the same mechanism it uses for parameters – it uses the bp and sp registers to access and allocate such variables. Consider the following Pascal program:

```
program LocalStorage;
var       i,j,k:integer;
          c: array [0..20000] of integer;

          procedure Proc1;
          var      a:array [0..30000] of integer;
                   i:integer;
          begin

          {Code that manipulates a and i}

          end;

          procedure Proc2;
          var      b:array [0..20000] of integer;
                   i:integer;
          begin

                   {Code that manipulates b and i}

          end;

begin

{main program that manipulates i,j,k, and c}

end.
```

Pascal normally allocates global variables in the data segment and local variables in the stack segment. Therefore, the program above allocates 50,002 words of local storage (30,001 words in Proc1 and 20,001 words in Proc2). This is above and beyond the other data on the stack (like return addresses). Since 50,002 words of storage consumes 100,004 bytes of storage you have a small problem – the 80x86 CPUs in real mode limit the stack segment to 65,536 bytes. Pascal avoids this problem by dynamically allocating local storage upon entering a procedure and deallocating local storage upon return. Unless Proc1 and Proc2 are both active (which can only occur if Proc1 calls Proc2 or vice versa), there is sufficient storage for this program. You don't need the 30,001 words for Proc1 and the 20,001 words for Proc2 at the same time. So Proc1 allocates and uses 60,002 bytes of storage, then deallocates this storage and returns (freeing up the 60,002 bytes). Next, Proc2 allocates 40,002 bytes of storage, uses them, deallocates them, and returns to its caller. Note that Proc1 and Proc2 share many of the same memory locations. However, they do so at different times. As long as these variables are temporaries whose values you needn't save from one invocation of the procedure to another, this form of local storage allocation works great.

The following comparison between a Pascal procedure and its corresponding assembly language code will give you a good idea of how to allocate local storage on the stack:

```
procedure LocalStuff(i,j,k:integer);
var l,m,n:integer; {local variables}
begin

       l := i+2;
       j := l*k+j;
       n := j-l;
       m := l+j+n;

end;
```

Calling sequence:

```
        LocalStuff(1,2,3);
```

Assembly language code:

```
LStuff_i        equu    8[bp]
LStuff_j        equ     6[bp]
LStuff_k        equ     4[bp]
LStuff_l        equ     -4[bp]
LStuff_m        equ     -6[bp]
LStuff_n        equ     -8[bp]

LocalStuff      proc    near
                push    bp
                mov     bp, sp
                push    ax
                sub     sp, 6                   ;Allocate local variables.
L0:             mov     ax, LStuff_i
                add     ax, 2
                mov     LStuff_l, ax
                mov     ax, LStuff_l
                mul     LStuff_k
                add     ax, LStuff_j
                mov     LStuff_j, ax
                sub     ax, LStuff_l            ;AX already contains j
                mov     LStuff_n, ax
                add     ax, LStuff_l            ;AX already contains n
                add     ax, LStuff_j
                mov     LStuff_m, ax

                add     sp, 6                   ;Deallocate local storage
                pop     ax
                pop     bp
                ret     6
LocalStuff      endp
```

The sub sp, 6 instruction makes room for three words on the stack. You can allocate l, m, and n in these three words. You can reference these variables by indexing off the bp register using negative offsets (see the code above). Upon reaching the statement at label L0, the stack looks something like Figure 11.15.

This code uses the matching add sp, 6 instruction at the end of the procedure to deallocate the local storage. The value you add to the stack pointer must exactly match the value you subtract when allocating this storage. If these two values don't match, the stack pointer upon entry to the routine will not match the stack pointer upon exit; this is like pushing or popping too many items inside the procedure.

Unlike parameters, that have a fixed offset in the activation record, you can allocate local variables in any order. As long as you are consistent with your location assignments, you can allocate them in any way you choose. Keep in mind, however, that the 80x86 supports two forms of the disp[bp] addressing mode. It uses a one byte displacement when it is in the range -128..+127. It uses a two byte displacement for values in the range -32,768..+32,767. Therefore, you should place all primitive data types and other small structures close to the base pointer, so you can use single byte displacements. You should place large arrays and other data structures below the smaller variables on the stack.

Most of the time you don't need to worry about allocating local variables on the stack. Most programs don't require more than 64K of storage. The CPU processes global variables faster than local variables. There are two situations where allocating local variables as globals in the data segment is not practical: when interfacing assembly language to HLLs like Pascal, and when writing recursive code. When interfacing to Pascal, your assembly language code may not have a data segment it can use, recursion often requires multiple instances of the same local variable.

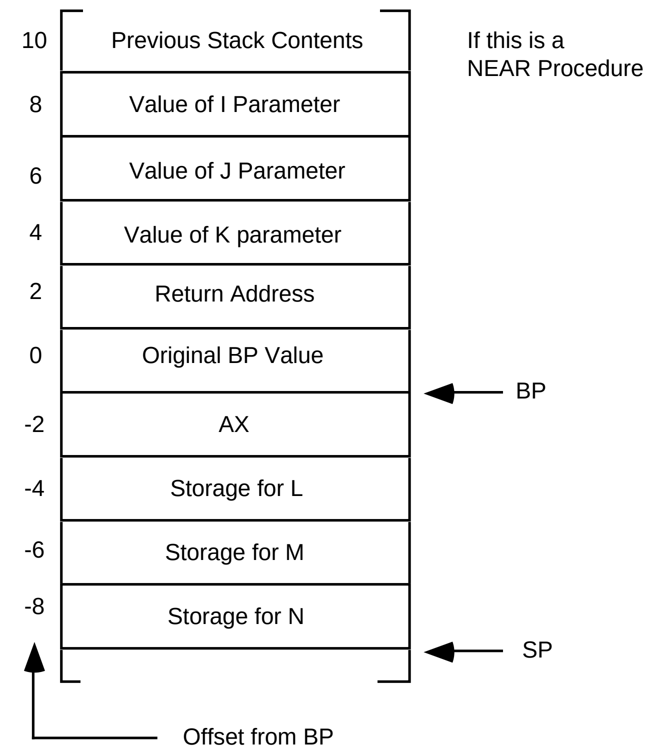

Figure 11.16 The Stack upon Entering the Next Procedure

11.9 Recursion

Recursion occurs when a procedure calls itself. The following, for example, is a recursive procedure:

```
Recursive       proc
                call     Recursive
                ret
Recursive       endp
```

Of course, the CPU will never execute the ret instruction at the end of this procedure. Upon entry into Recursive, this procedure will immediately call itself again and control will never pass to the ret instruction. In this particular case, run away recursion results in an infinite loop.

In many respects, recursion is very similar to iteration (that is, the repetitive execution of a loop). The following code also produces an infinite loop:

```
Recursive       proc
                jmp      Recursive
                ret
Recursive       endp
```

There is, however, one major difference between these two implementations. The former version of Recursive pushes a return address onto the stack with each invocation of the subroutine. This does not happen in the example immediately above (since the jmp instruction does not affect the stack).

Like a looping structure, recursion requires a termination condition in order to stop infinite recursion. Recursive could be rewritten with a termination condition as follows:

```
Recursive        proc
                 dec      ax
                 jz       QuitRecurse
                 call     Recursive
QuitRecurse:     ret
Recursive        endp
```

This modification to the routine causes Recursive to call itself the number of times appearing in the ax register. On each call, Recursive decrements the ax register by one and calls itself again. Eventually, Recursive decrements ax to zero and returns. Once this happens, the CPU executes a string of ret instructions until control returns to the original call to Recursive.

So far, however, there hasn't been a real need for recursion. After all, you could efficiently code this procedure as follows:

```
Recursive        proc
RepeatAgain:     dec      ax
                 jnz      RepeatAgain
                 ret
Recursive        endp
```

Both examples would repeat the body of the procedure the number of times passed in the ax register[9]. As it turns out, there are only a few recursive algorithms that you cannot implement in an iterative fashion. However, many recursively implemented algorithms are more efficient than their iterative counterparts and most of the time the recursive form of the algorithm is much easier to understand.

The quicksort algorithm is probably the most famous algorithm that almost always appears in recursive form. A Pascal implementation of this algorithm follows:

```
procedure quicksort(var a:ArrayToSort; Low,High: integer);

       procedure sort(l,r: integer);
       var i,j,Middle,Temp: integer;
       begin

                 i:=l;
                 j:=r;
                 Middle:=a[(l+r) DIV 2];
                 repeat
                          while (a[i] < Middle) do i:=i+1;
                          while (Middle < a[j]) do j:=j-1;
                          if (i <= j) then begin

                            Temp:=a[i];
                            a[i]:=a[j];
                            a[j]:=Temp;
                            i:=i+1;
                            j:=j-1;

                          end;

                 until i>j;
                 if l<j then sort(l,j);
                 if i<r then sort(i,r);

       end;

begin {quicksort};

       sort(Low,High);

end;
```

The sort subroutine is the recursive routine in this package. Recursion occurs at the last two if statements in the sort procedure.

In assembly language, the sort routine looks something like this:

9. Although the latter version will do it considerably faster since it doesn't have the overhead of the CALL/RET instructions.

```
                include     stdlib.a
                includelib  stdlib.lib
cseg            segment
                assume      cs:cseg, ds:cseg, ss:sseg, es:cseg

; Main program to test sorting routine

Main            proc
                mov     ax, cs
                mov     ds, ax
                mov     es, ax

                mov     ax, 0
                push    ax
                mov     ax, 31
                push    ax
                call    sort

                ExitPgm                         ;Return to DOS
Main            endp

; Data to be sorted

a               word    31,30,29,28,27,26,25,24,23,22,21,20,19,18,17,16
                word    15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0

; procedure sort (l,r:integer)
; Sorts array A between indices l and r

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

sort            proc    near
                push    bp
                mov     bp, sp
                sub     sp, 4                   ;Make room for i and j.

                mov     ax, l                   ;i := l
                mov     i, ax
                mov     bx, r                   ;j := r

                mov     j, bx

; Note: This computation of the address of a[(l+r) div 2] is kind
; of strange. Rather than divide by two, then multiply by two
; (since A is a word array), this code simply clears the L.O. bit
; of BX.

                add     bx, l                   ;Middle := a[(l+r) div 2]
                and     bx, 0FFFEh
                mov     ax, a[bx]               ;BX*2, because this is a word
;                                               ; array,nullifies the “div 2”
;                                               ; above.
;
; Repeat until i > j: Of course, I and J are in BX and SI.

                lea     bx, a                   ;Compute the address of a[i]
                add     bx, i                   ; and leave it in BX.
                add     bx, i

                lea     si, a                   ;Compute the address of a[j]
                add     si, j                   ; and leave it in SI.
                add     si, j

RptLp:

; While (a [i] < Middle) do i := i + 1;

                sub     bx, 2                   ;We’ll increment it real
soon.
WhlLp1:         add     bx, 2
                cmp     ax, [bx]                ;AX still contains middle
                jg      WhlLp1

; While (Middle < a[j]) do j := j-1
```

```
                add     si, 2                   ;We'll decrement it in loop
WhlLp2:         add     si, 2
                cmp     ax, [si]                ;AX still contains middle
                jl      WhlLp2                  ; value.
                cmp     bx, si
                jnle    SkipIf

; Swap, if necessary

                mov     dx, [bx]
                xchg    dx, [si]
                xchg    dx, [bx]

                add     bx, 2                   ;Bump by two (integer values)
                sub     si, 2

SkipIf:         cmp     bx, si
                jng     RptLp

; Convert SI and BX back to I and J

                lea     ax, a
                sub     bx, ax
                shr     bx, 1
                sub     si, ax
                        shrsi, 1

; Now for the recursive part:

                mov     ax, l
                cmp     ax, si
                jnl     NoRec1
                push    ax
                push    si
                call    sort

NoRec1:         cmp     bx, r
                jnl     NoRec2
                push    bx
                push    r
                call    sort
NoRec2:         mov     sp, bp
                pop     bp
                ret     4

Sort            endp

cseg            ends
sseg            segment stack 'stack'
                word    256 dup (?)
sseg            ends
                end     main
```

Other than some basic optimizations (like keeping several variables in registers), this code is almost a literal translation of the Pascal code. Note that the local variables i and j aren't necessary in this assembly language code (we could use registers to hold their values). Their use simply demonstrates the allocation of local variables on the stack.

There is one thing you should keep in mind when using recursion – recursive routines can eat up a considerable stack space. Therefore, when writing recursive subroutines, always allocate sufficient memory in your stack segment. The example above has an extremely anemic 512 byte stack space, however, it only sorts 32 numbers therefore a 512 byte stack is sufficient. In general, you won't know the depth to which recursion will take you, so allocating a large block of memory for the stack may be appropriate.

There are several efficiency considerations that apply to recursive procedures. For example, the second (recursive) call to sort in the assembly language code above need not be a recursive call. By setting up a couple of variables and registers, a simple jmp instruction can can replace the pushes and the recursive call. This will improve the performance of the quicksort routine (quite a bit, actually) and will reduce the amount of memory the stack requires. A good book on algorithms, such as D.E. Knuth's The Art of Computer Programming, Volume 3, would be an excellent source of additional material on quicksort. Other texts on algorithm complexity, recursion theory, and algorithms would be a good place to look for ideas on efficiently implementing recursive algorithms.

11.10 Sample Program

The following sample program demonstrates several concepts appearing in this chapter, most notably, passing parameters on the stack. This program (Pgm11_1.asm appearing on the companion CD-ROM) manipulates the PC's memory-mapped text video display screen (at address B800:0 for color displays, B000:0 for monochrome displays). It provides routines that "capture" all the data on the screen to an array, write the contents of an array to the screen, clear the screen, scroll one line up or down, position the cursor at an (X,Y) coordinate, and retrieve the current cursor position.

Note that this code was written to demonstrate the use of parameters and local variables. Therefore, it is rather inefficient. As the comments point out, many of the functions this package provides could be written to run much faster using the 80x86 string instructions. See the laboratory exercises for a different version of some of these functions that is written in such a fashion. Also note that this code makes some calls to the PC's BIOS to set and obtain the cursor position as well as clear the screen. See the chapter on BIOS and DOS for more details on these BIOS calls.

```
; Pgm11_1.asm
;
; Screen Aids.
;
; This program provides some useful screen manipulation routines
; that let you do things like position the cursor, save and restore
; the contents of the display screen, clear the screen, etc.
;
; This program is not very efficient.  It was written to demonstrate
; parameter passing, use of local variables, and direct conversion of
; loops to assembly language.  There are far better ways of doing
; what this program does (running about 5-10x faster) using the 80x86
; string instructions.



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

                 .386                   ;Comment out these two statements
                 option   segment:use16 ; if you are not using an 80386.


; ScrSeg- This is the video screen's segment address.  It should be
;                B000 for mono screens and B800 for color screens.

ScrSeg           =        0B800h

dseg             segment  para public 'data'

XPosn            word     ?            ;Cursor X-Coordinate (0..79)
YPosn            word     ?            ;Cursor Y-Coordinate (0..24)

; The following array holds a copy of the initial screen data.

SaveScr          word     25 dup (80 dup (?))

dseg             ends


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

```
; Capture-        Copies the data on the screen to the array passed
;                 by reference as a parameter.
;
; procedure Capture(var ScrCopy:array[0..24,0..79] of word);
; var x,y:integer;
; begin
;
;      for y := 0 to 24 do
;          for x := 0 to 79 do
;                SCREEN[y,x] := ScrCopy[y,x];
; end;
;
;
; Activation record for Capture:
;
;      |                        |
;      | Previous stk contents |
;      -------------------------
;      |  ScrCopy Seg Adrs      |
;      --                      --
;      | ScrCopy offset Adrs    |
;      -------------------------
;      | Return Adrs (near)     |
;      -------------------------
;      |      Old BP            |
;      ------------------------- <- BP
;      |  X coordinate value    |
;      -------------------------
;      |  Y coordinate value    |
;      -------------------------
;      | Registers, etc.        |
;      ------------------------- <- SP


ScrCopy_cap      textequ  <dword ptr [bp+4]>
X_cap            textequ  <word ptr [bp-2]>
Y_cap            textequ  <word ptr [bp-4]>

Capture          proc
                 push     bp
                 mov      bp, sp
                 sub      sp, 4         ;Allocate room for locals.

                 push     es
                 push     ds
                 push     ax
                 push     bx
                 push     di

                 mov      bx, ScrSeg    ;Set up pointer to SCREEN
                 mov      es, bx        ; memory (ScrSeg:0).

                 lds      di, ScrCopy_cap ;Get ptr to capture array.

                 mov      Y_cap, 0
YLoop:           mov      X_cap, 0
XLoop:           mov      bx, Y_cap
                 imul     bx, 80        ;Screen memory is a 25x80 array
                 add      bx, X_cap     ; stored in row major order
                 add      bx, bx        ; with two bytes per element.

                 mov      ax, es:[bx]   ;Read character code from screen.
                 mov      [di][bx], ax ;Store away into capture array.

                 inc      X_Cap         ;Repeat for each character on this
                 cmp      X_Cap, 80     ; row of characters (each character
                 jb       XLoop         ; in the row is two bytes).

                 inc      Y_Cap         ;Repeat for each row on the screen.
```

```
                cmp     Y_Cap, 25
                jb      YLoop

                pop     di
                pop     bx
                pop     ax
                pop     ds
                pop     es
                mov     sp, bp
                pop     bp
                ret     4
Capture         endp


; Fill-         Copies array passed by reference onto the screen.
;
; procedure Fill(var ScrCopy:array[0..24,0..79] of word);
; var x,y:integer;
; begin
;
;       for y := 0 to 24 do
;           for x := 0 to 79 do
;                 ScrCopy[y,x] := SCREEN[y,x];
; end;
;
;
; Activation record for Fill:
;
;       |                         |
;       | Previous stk contents |
;       -------------------------
;       |  ScrCopy Seg Adrs     |
;       --                     --
;       | ScrCopy offset Adrs   |
;       -------------------------
;       | Return Adrs (near)    |
;       -------------------------
;       |      Old BP           |
;       ------------------------- <- BP
;       |  X coordinate value   |
;       -------------------------
;       |  Y coordinate value   |
;       -------------------------
;       | Registers, etc.       |
;       ------------------------- <- SP



ScrCopy_fill    textequ  <dword ptr [bp+4]>
X_fill          textequ  <word ptr [bp-2]>
Y_fill          textequ  <word ptr [bp-4]>

Fill            proc
                push    bp
                mov     bp, sp
                sub     sp, 4

                push    es
                push    ds
                push    ax
                push    bx
                push    di

                mov     bx, ScrSeg    ;Set up pointer to SCREEN
                mov     es, bx        ; memory (ScrSeg:0).

                lds     di, ScrCopy_fill ;Get ptr to data array.

                mov     Y_Fill, 0
YLoop:          mov     X_Fill, 0
```

```
XLoop:          mov     bx, Y_Fill
                imul    bx, 80       ;Screen memory is a 25x80 array
                add     bx, X_Fill   ; stored in row major order
                add     bx, bx       ; with two bytes per element.

                mov     ax, [di][bx] ;Store away into capture array.
                mov     es:[bx], ax  ;Read character code from screen.

                inc     X_Fill       ;Repeat for each character on this
                cmp     X_Fill, 80   ; row of characters (each character
                jb      XLoop        ; in the row is two bytes).

                inc     Y_Fill       ;Repeat for each row on the screen.
                cmp     Y_Fill, 25
                jb      YLoop

                pop     di
                pop     bx
                pop     ax
                pop     ds
                pop     es
                mov     sp, bp
                pop     bp
                ret     4
Fill            endp


; Scroll_up-    Scrolls the screen up on line.  It does this by copying the
;               second line to the first, the third line to the second, the
;               fourth line to the third, etc.
;
; procedure Scroll_up;
; var x,y:integer;
; begin
;       for y := 1 to 24 do
;           for x := 0 to 79 do
;               SCREEN[Y-1,X] := SCREEN[Y,X];
; end;
;
; Activation record for Scroll_up:
;
;       |                        |
;       | Previous stk contents  |
;       -------------------------
;       | Return Adrs (near)     |
;       -------------------------
;       |     Old BP             |
;       ------------------------- <- BP
;       |  X coordinate value    |
;       -------------------------
;       |  Y coordinate value    |
;       -------------------------
;       | Registers, etc.        |
;       ------------------------- <- SP


X_su            textequ  <word ptr [bp-2]>
Y_su            textequ  <word ptr [bp-4]>

Scroll_up       proc
                push    bp
                mov     bp, sp
                sub     sp, 4        ;Make room for X, Y.

                push    ds
                push    ax
                push    bx

                mov     ax, ScrSeg
                mov     ds, ax
```

```
                mov     Y_su, 0
su_Loop1:       mov     X_su, 0

su_Loop2:       mov     bx, Y_su     ;Compute index into screen
                imul    bx, 80       ; array.
                add     bx, X_su
                add     bx, bx       ;Remember, this is a word array.

                mov     ax, [bx+160] ;Fetch word from source line.
                mov     [bx], ax     ;Store into dest line.

                inc     X_su
                cmp     X_su, 80
                jb      su_Loop2

                inc     Y_su
                cmp     Y_su, 80
                jb      su_Loop1

                pop     bx
                pop     ax
                pop     ds
                mov     sp, bp
                pop     bp
                ret
Scroll_up       endp


; Scroll_dn-    Scrolls the screen down one line.  It does this by copying the
;               24th line to the 25th, the 23rd line to the 24th, the 22nd line
;               to the 23rd, etc.
;
; procedure Scroll_dn;
; var x,y:integer;
; begin
;       for y := 23 downto 0 do
;           for x := 0 to 79 do
;                 SCREEN[Y+1,X] := SCREEN[Y,X];
; end;
;
; Activation record for Scroll_dn:
;
;       |                        |
;       | Previous stk contents  |
;       -------------------------
;       | Return Adrs (near)     |
;       -------------------------
;       |      Old BP            |
;       ------------------------- <- BP
;       |  X coordinate value    |
;       -------------------------
;       |  Y coordinate value    |
;       -------------------------
;       | Registers, etc.        |
;       ------------------------- <- SP


X_sd            textequ <word ptr [bp-2]>
Y_sd            textequ <word ptr [bp-4]>

Scroll_dn       proc
                push    bp
                mov     bp, sp
                sub     sp, 4        ;Make room for X, Y.

                push    ds
                push    ax
                push    bx

                mov     ax, ScrSeg
```

```
                mov     ds, ax
                mov     Y_sd, 23
sd_Loop1:       mov     X_sd, 0

sd_Loop2:       mov     bx, Y_sd     ;Compute index into screen
                imul    bx, 80       ; array.
                add     bx, X_sd
                add     bx, bx       ;Remember, this is a word array.

                mov     ax, [bx]     ;Fetch word from source line.
                mov     [bx+160], ax ;Store into dest line.

                inc     X_sd
                cmp     X_sd, 80
                jb      sd_Loop2

                dec     Y_sd
                cmp     Y_sd, 0
                jge     sd_Loop1

                pop     bx
                pop     ax
                pop     ds
                mov     sp, bp
                pop     bp
                ret
Scroll_dn       endp



; GotoXY-       Positions the cursor at the specified X, Y coordinate.
;
; procedure gotoxy(x,y:integer);
; begin
;       BIOS(posnCursor,x,y);
; end;
;
; Activation record for GotoXY
;
;       |                        |
;       | Previous stk contents |
;       -------------------------
;       |  X coordinate value   |
;       -------------------------
;       |  Y coordinate value   |
;       -------------------------
;       | Return Adrs (near)    |
;       -------------------------
;       |      Old BP           |
;       ------------------------- <- BP
;       | Registers, etc.       |
;       ------------------------- <- SP


X_gxy           textequ  <byte ptr [bp+6]>
Y_gxy           textequ  <byte ptr [bp+4]>

GotoXY          proc
                push    bp
                mov     bp, sp
                push    ax
                push    bx
                push    dx

                mov     ah, 2        ;Magic BIOS value for gotoxy.
                mov     bh, 0        ;Display page zero.
                mov     dh, Y_gxy    ;Set up BIOS (X,Y) parameters.
                mov     dl, X_gxy
                int     10h          ;Make the BIOS call.
```

```
                pop     dx
                pop     bx
                pop     ax
                mov     sp, bp
                pop     bp
                ret     4
GotoXY          endp


; GetX-         Returns cursor X-Coordinate in the AX register.

GetX            proc
                push    bx
                push    cx
                push    dx

                mov     ah, 3           ;Read X, Y coordinates from
                mov     bh, 0           ; BIOS
                int     10h

                mov     al, dl          ;Return X coordinate in AX.
                mov     ah, 0

                pop     dx
                pop     cx
                pop     bx
                ret
GetX            endp



; GetY-         Returns cursor Y-Coordinate in the AX register.

GetY            proc
                push    bx
                push    cx
                push    dx

                mov     ah, 3
                mov     bh, 0
                int     10h

                mov     al, dh          ;Return Y Coordinate in AX.
                mov     ah, 0

                pop     dx
                pop     cx
                pop     bx
                ret
GetY            endp



; ClearScrn-    Clears the screen and positions the cursor at (0,0).
;
; procedure ClearScrn;
; begin
;       BIOS(Initialize)
; end;

ClearScrn       proc
                push    ax
                push    bx
                push    cx
                push    dx

                mov     ah, 6           ;Magic BIOS number.
                mov     al, 0           ;Clear entire screen.
                mov     bh, 07          ;Clear with black spaces.
```

```
                mov     cx, 0000;Upper left corner is (0,0)
                mov     dl, 79          ;Lower X-coordinate
                mov     dh, 24          ;Lower Y-coordinate
                int     10h             ;Make the BIOS call.

                push    0               ;Position the cursor to (0,0)
                push    0               ; after the call.
                call    GotoXY

                pop     dx
                pop     cx
                pop     bx
                pop     ax
                ret
ClearScrn       endp


; A short main program to test out the above:

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

; Save the screen as it looks when this program is run.

                push    seg SaveScr
                push    offset SaveScr
                call    Capture

                call    GetX
                mov     XPosn, ax

                call    GetY
                mov     YPosn, ax


; Clear the screen to prepare for our stuff.

                call    ClearScrn

; Position the cursor in the middle of the screen and print some stuff.

                push    30              ;X value
                push    10              ;Y value
                call    GotoXY

                print
                byte    "Screen Manipulatation Demo",0

                push    30
                push    11
                call    GotoXY

                print
                byte    "Press any key to continue",0

                getc


; Scroll the screen up two lines

                call    Scroll_up
                call    Scroll_up
                getc

;Scroll the screen down four lines:

                call    Scroll_dn
```

```
                 call     Scroll_dn
                 call     Scroll_dn
                 call     Scroll_dn
                 getc

; Restore the screen to what it looked like prior to this call.

                 push     seg SaveScr
                 push     offset SaveScr
                 call     Fill

                 push     XPosn
                 push     YPosn
                 call     GotoXY

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        byte     16 dup (?)
zzzzzzseg        ends
                 end      Main
```

11.11 Laboratory Exercises

This laboratory exercise demonstrates how a C/C++ program calls some assembly language functions. This exercise consists of two program units: a Borland C++ program (Ex11_1.cpp) and a MASM 6.11 program (Ex11_1a.asm). Since you may not have access to a C++ compiler (and Borland C++ in particular)[10], the EX11.EXE file contains a precompiled and linked version of these files. If you have a copy of Borland C++ then you can compile/assemble these files using the makefile that also appears in the Chapter 11 subdirectory.

The C++ program listing appears in Section 11.11.1. This program clears the screen and then bounces a pound sign ("#") around the screen until the user presses any key. Then this program restores the screen to the previous display before running the program and quits. All screen manipulation, as well as testing for a keypress, is taken care of by functions written in assembly language. The "extern" statements at the beginning of the program provide the linkage to these assembly language functions[11]. There are a few important things to note about how C/C++ passes parameters to an assembly language function:

- C++ pushes parameters on the stack in the *reverse* order that they appear in a parameter list. For example, for the call "f(a,b);" C++ would push b first and a second. This is opposite of most of the examples in this chapter.
- In C++, the caller is responsible for removing parameters from the stack. In this chapter, the callee (the function itself) usually removed the parameters by specifying some value after the ret instruction. Assembly language functions that C++ calls must *not* do this.
- C++ on the PC uses different memory models to control whether pointers and functions are near or far. This particular program uses the *compact*

10. There is nothing Borland specific in this C++ program. Borland was chosen because it provides an option that generates well annotated assembly output.

11. The *extern "C"* phrase instructs Borland C++ to generate standard C external names rather than C++ *mangled* names. A C external name is the function name with an underscore in front of it (e.g., GotoXY becomes _GotoXY). C++ completely changes the name to handle overloading and it is difficult to predict the actual name of the corresponding assembly language function.

memory model. This provides for near procedures and far pointers. Therefore, all calls will be near (with only a two-byte return address on the stack) and all pointers to data objects will be far.

- Borland C++ requires a function to preserve the segment registers, BP, DI, and SI. The function need not preserve any other registers. If an assembly language function needs to return a 16-bit function result to C++, it must return this value in the AX register.
- See the Borland C++ Programmer's Guide (or corresponding manual for your C++ compiler) for more details about the C/C++ and assembly language interface.

Most C++ compilers give you the option of generating assembly language output rather than binary machine code. Borland C++ is nice because it generates nicely annotated assembly output with comments pointing out which C++ statments correspond to a given sequence of assembly language instructions. The assembly language output of BCC appears in Section 11.11.2 (This is a slightly edited version to remove some superfluous information). Look over this code and note that, subject to the rules above, the C++ compiler emits code that is very similar to that described throughout this chapter.

The Ex11_1a.asm file (see section 11.11.3) is the actual assembly code the C++ program calls. This contains the functions for the GotoXY, GetXY, ClrScrn, tstKbd, Capture, PutScrn, PutChar, and PutStr routines that Ex11_1.cpp calls. To avoid legal software distribution problems, this particular C/C++ program does not include any calls to C/C++ Standard Library functions. Furthermore, it does not use the standard C0m.obj file from Borland that calls the main program. Borland's liberal license agreement does *not* allow one to distribute their librarys and object modules unlinked with other code. The assembly language code provides the necessary I/O routines and it also provides a startup routine (StartPgm) that call the C++ main program when DOS/Windows transfers control to the program. By supplying the routines this way, you do not need the Borland libraries or object code to link these programs together.

One side effect of linking the modules in this fashion is that the compiler, assembler, and linker cannot store the correct source level debugging information in the .exe file. Therefore, you will not be able to use CodeView to view the actual source code. Instead, you will have to work with disassembled machine code. This is where the assembly output from Borland C++ (Ex11_1.asm) comes in handy. As you single step through the main C++ program you can trace the program flow by looking at the Ex11_1.asm file.

For your lab report: single step through the StartPgm code until it calls the C++ main function. When this happens, locate the calls to the routines in Ex11_1a.asm. Set breakpoints on each of these calls using the F9 key. Run up to each breakpoint and then single step into the function using the F8 key. Once inside, display the memory locations starting at SS:SP. Identify each parameter passed on the stack. For reference parameters, you may want to look at the memory locations whose address appears on the stack. Report your findings in your lab report.

Include a printout of the Ex11_1.asm file and identify those instructions that push each parameter onto the stack. At run time, determine the values that each parameter push sequence pushes onto the stack and include these values in your lab report.

Many of the functions in the assembly file take a considerable amount of time to execute. Therefore, you should not single step through each of the functions. Instead, make sure you've set up the breakpoints on each of the call instructions in the C++ program and use the F5 key to run (at full speed) up to the next function call.

11.11.1 Ex11_1.cpp

```
extern "C" void GotoXY(unsigned y, unsigned x);
extern "C" void GetXY(unsigned &x, unsigned &y);
extern "C" void ClrScrn();
extern "C" int tstKbd();
```

```
extern "C" void Capture(unsigned ScrCopy[25][80]);
extern "C" void PutScr(unsigned ScrCopy[25][80]);
extern "C" void PutChar(char ch);
extern "C" void PutStr(char *ch);

int main()
{
    unsigned SaveScr[25][80];

    int         dx,
                x,
                dy,
                y;

    long        i;

    unsigned    savex,
                savey;



    GetXY(savex, savey);
    Capture(SaveScr);
    ClrScrn();

    GotoXY(24,0);
    PutStr("Press any key to quit");

    dx = 1;
    dy = 1;
    x = 1;
    y = 1;
    while (!tstKbd())
    {

       GotoXY(y, x);
        PutChar('#');

        for (i=0; i<500000; ++i);

        GotoXY(y, x);
        PutChar(' ');

       x += dx;
        y += dy;
        if (x >= 79)
       {
            x = 78;
            dx = -1;
       }
        else if (x <= 0)
       {
            x = 1;
            dx = 1;
       }

        if (y >= 24)
       {
            y = 23;
            dy = -1;
       }
        else if (y <= 0)
       {
            y = 1;
            dy = 1;
       }

    }

    PutScr(SaveScr);
```

```
    GotoXY(savey, savex);
    return 0;
}
```

11.11.2 Ex11_1.asm

```
_TEXT  segment byte public 'CODE'
_TEXT  ends
DGROUP group     _DATA,_BSS
       assume    cs:_TEXT,ds:DGROUP
_DATA  segment word public 'DATA'
d@     label     byte
d@w    label     word
_DATA  ends
_BSS   segment word public 'BSS'
b@     label     byte
b@w    label     word
_BSS   ends

_TEXT  segment byte public 'CODE'
   ;
   ;   int main()
   ;
       assume    cs:_TEXT
_main  proc      near
       push      bp
       mov       bp,sp
       sub       sp,4012
       push      si
       push      di
   ;
   ;   {
   ;       unsigned SaveScr[25][80];
   ;
   ;       int         dx,
   ;                   x,
   ;                   dy,
   ;                   y;
   ;
   ;       long        i;
   ;
   ;       unsigned    savex,
   ;                   savey;
   ;
   ;
   ;
   ;       GetXY(savex, savey);
   ;
       push      ss
       lea       ax,word ptr [bp-12]
       push      ax
       push      ss
       lea       ax,word ptr [bp-10]
       push      ax
       call      near ptr _GetXY
       add       sp,8
   ;
   ;       Capture(SaveScr);
   ;
       push      ss
       lea       ax,word ptr [bp-4012]
       push      ax
       call      near ptr _Capture
       pop       cx
       pop       cx
   ;
```

```
    ;       ClrScrn();
    ;
        call      near ptr _ClrScrn
    ;
    ;
    ;       GotoXY(24,0);
    ;
        xor       ax,ax
        push      ax
        mov       ax,24
        push      ax
        call      near ptr _GotoXY
        pop       cx
        pop       cx
    ;
    ;       PutStr("Press any key to quit");
    ;
        push      ds
        mov       ax,offset DGROUP:s@
        push      ax
        call      near ptr _PutStr
        pop       cx
        pop       cx
    ;
    ;
    ;       dx = 1;
    ;
        mov       word ptr [bp-2],1
    ;
    ;       dy = 1;
    ;
        mov       word ptr [bp-4],1
    ;
    ;       x = 1;
    ;
        mov       si,1
    ;
    ;       y = 1;
    ;
        mov       di,1
        jmp       @1@422
@1@58:
    ;
    ;       while (!tstKbd())
    ;       {
    ;
    ;             GotoXY(y, x);
    ;
        push      si
        push      di
        call      near ptr _GotoXY
        pop       cx
        pop       cx
    ;
    ;           PutChar('#');
    ;
        mov       al,35
        push      ax
        call      near ptr _PutChar
        pop       cx
    ;
    ;
    ;           for (i=0; i<500000; ++i);
    ;
        mov       word ptr [bp-6],0
        mov       word ptr [bp-8],0
        jmp       short @1@114
@1@86:
        add       word ptr [bp-8],1
        adc       word ptr [bp-6],0
```

```
@1@114:
        cmp       word ptr [bp-6],7
        jl        short @1@86
        jne       short @1@198
        cmp       word ptr [bp-8],-24288
        jb        short @1@86
@1@198:
   ;
   ;
   ;            GotoXY(y, x);
   ;
        push      si
        push      di
        call      near ptr _GotoXY
        pop       cx
        pop       cx
   ;
   ;            PutChar(' ');
   ;
        mov       al,32
        push      ax
        call      near ptr _PutChar
        pop       cx
   ;
   ;
   ;
   ;
   ;              x += dx;
   ;
        add       si,word ptr [bp-2]
   ;
   ;            y += dy;
   ;
        add       di,word ptr [bp-4]
   ;
   ;            if (x >= 79)
   ;
        cmp       si,79
        jl        short @1@254
   ;
   ;              {
   ;                x = 78;
   ;
        mov       si,78
   ;
   ;                dx = -1;
   ;
        mov       word ptr [bp-2],-1
   ;
   ;              }
   ;
        jmp       short @1@310
@1@254:
   ;
   ;            else if (x <= 0)
   ;
        or        si,si
        jg        short @1@310
   ;
   ;              {
   ;                x = 1;
   ;
        mov       si,1
   ;
   ;                dx = 1;
   ;
        mov       word ptr [bp-2],1
@1@310:
   ;
   ;              }
```

```
   ;
   ;          if (y >= 24)
   ;
        cmp       di,24
        jl        short @1@366
   ;
   ;            {
   ;              y = 23;
   ;
        mov       di,23
   ;
   ;              dy = -1;
   ;
        mov       word ptr [bp-4],-1
   ;
   ;            }
   ;
        jmp       short @1@422
@1@366:
   ;
   ;          else if (y <= 0)
   ;
        or        di,di
        jg        short @1@422
   ;
   ;            {
   ;              y = 1;
   ;
        mov       di,1
   ;
   ;              dy = 1;
   ;
        mov       word ptr [bp-4],1
@1@422:
        call      near ptr _tstKbd
        or        ax,ax
        jne       @@0
        jmp       @1@58
@@0:
   ;
   ;            }
   ;
   ;
   ;      }
   ;
   ;      PutScr(SaveScr);
   ;
        push      ss
        lea       ax,word ptr [bp-4012]
        push      ax
        call      near ptr _PutScr
        pop       cx
        pop       cx
   ;
   ;      GotoXY(savey, savex);
   ;
        push      word ptr [bp-10]
        push      word ptr [bp-12]
        call      near ptr _GotoXY
        pop       cx
        pop       cx
   ;
   ;      return 0;
   ;
        xor       ax,ax
        jmp       short @1@478
@1@478:
   ;
   ;   }
   ;
```

```
        pop       di
        pop       si
        mov       sp,bp
        pop       bp
        ret
_main   endp

_TEXT   ends

_DATA   segment word public 'DATA'
s@      label     byte
        db        'Press any key to quit'
        db        0
_DATA   ends
_TEXT   segment byte public 'CODE'
_TEXT   ends
        public    _main
        extrn     _PutStr:near
        extrn     _PutChar:near
        extrn     _PutScr:near
        extrn     _Capture:near
        extrn     _tstKbd:near
        extrn     _ClrScrn:near
        extrn     _GetXY:near
        extrn     _GotoXY:near
_s@     equ       s@
        end
```

11.11.3 EX11_1a.asm

```
; Assembly code to link with a C/C++ program.
; This code directly manipulates the screen giving C++
; direct access control of the screen.
;
; Note: Like PGM11_1.ASM, this code is relatively inefficient.
; It could be sped up quite a bit using the 80x86 string instructions.
; However, its inefficiency is actually a plus here since we don't
; want the C/C++ program (Ex11_1.cpp) running too fast anyway.
;
;
; This code assumes that Ex11_1.cpp is compiled using the LARGE
; memory model (far procs and far pointers).

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

                 .386                   ;Comment out these two statements
                 option   segment:use16 ; if you are not using an 80386.

; ScrSeg- This is the video screen's segment address.  It should be
;                B000 for mono screens and B800 for color screens.

ScrSeg           =        0B800h

_TEXT           segment para public 'CODE'
                assume  cs:_TEXT

; _Capture-      Copies the data on the screen to the array passed
;                by reference as a parameter.
;
; procedure Capture(var ScrCopy:array[0..24,0..79] of word);
; var x,y:integer;
; begin
;
;      for y := 0 to 24 do
;          for x := 0 to 79 do
```

```
;                 SCREEN[y,x] := ScrCopy[y,x];
; end;
;
;
; Activation record for Capture:
;
;       |                        |
;       | Previous stk contents |
;       -------------------------
;       |  ScrCopy Seg Adrs      |
;       --                      --
;       | ScrCopy offset Adrs    |
;       -------------------------
;       | Return Adrs (offset)   |
;       -------------------------
;       |  X coordinate value    |
;       -------------------------
;       |  Y coordinate value    |
;       -------------------------
;       | Registers, etc.        |
;       ------------------------- <- SP


ScrCopy_cap      textequ  <dword ptr [bp+4]>
X_cap            textequ  <word ptr [bp-2]>
Y_cap            textequ  <word ptr [bp-4]>

                 public   _Capture
_Capture         proc     near
                 push     bp
                 mov      bp, sp

                 push     es
                 push     ds
                 push     si
                 push     di
                 pushf
                 cld

                 mov      si, ScrSeg              ;Set up pointer to SCREEN
                 mov      ds, si                  ; memory (ScrSeg:0).
                 sub      si, si

                 les      di, ScrCopy_cap         ;Get ptr to capture array.

                 mov      cx, 1000                ;4000 dwords on the screen
        rep      movsd

                 popf
                 pop      di
                 pop      si
                 pop      ds
                 pop      es
                 mov      sp, bp
                 pop      bp
                 ret
_Capture         endp


; _PutScr-      Copies array passed by reference onto the screen.
;
; procedure PutScr(var ScrCopy:array[0..24,0..79] of word);
; var x,y:integer;
; begin
;
;       for y := 0 to 24 do
;           for x := 0 to 79 do
;                 ScrCopy[y,x] := SCREEN[y,x];
; end;
;
```

```
;
; Activation record for PutScr:
;
;       |                        |
;       | Previous stk contents |
;       -------------------------
;       |  ScrCopy Seg Adrs      |
;       --                      --
;       | ScrCopy offset Adrs   |
;       -------------------------
;       | Return Adrs (offset)  |
;       -------------------------
;       |     BP Value          | <- BP
;       -------------------------
;       |  X coordinate value   |
;       -------------------------
;       |  Y coordinate value   |
;       -------------------------
;       | Registers, etc.       |
;       ------------------------- <- SP


ScrCopy_fill    textequ  <dword ptr [bp+4]>
X_fill          textequ  <word ptr [bp-2]>
Y_fill          textequ  <word ptr [bp-4]>

                public  _PutScr
_PutScr         proc    near
                push     bp
                mov      bp, sp

                push     es
                push     ds
                push     si
                push     di
                pushf
                cld

                mov      di, ScrSeg             ;Set up pointer to SCREEN
                mov      es, di                 ; memory (ScrSeg:0).
                sub      di, di

                lds      si, ScrCopy_cap        ;Get ptr to capture array.

                mov      cx, 1000               ;1000 dwords on the screen
        rep     movsd

                popf
                pop      di
                pop      si
                pop      ds
                pop      es
                mov      sp, bp
                pop      bp
                ret
_PutScr         endp


; GotoXY-Positions the cursor at the specified X, Y coordinate.
;
; procedure gotoxy(y,x:integer);
; begin
;       BIOS(posnCursor,x,y);
; end;
;
; Activation record for GotoXY
;
```

```
;       |                        |
;       | Previous stk contents |
;       -------------------------
;       |  X coordinate value   |
;       -------------------------
;       |  Y coordinate value   |
;       -------------------------
;       |  Return Adrs (offset) |
;       -------------------------
;       |      Old BP           |
;       ------------------------- <- BP
;       | Registers, etc.       |
;       ------------------------- <- SP


X_gxy           textequ <byte ptr [bp+6]>
Y_gxy           textequ <byte ptr [bp+4]>

                public   _GotoXY
_GotoXY         proc     near
                push     bp
                mov      bp, sp

                mov      ah, 2         ;Magic BIOS value for gotoxy.
                mov      bh, 0         ;Display page zero.
                mov      dh, Y_gxy     ;Set up BIOS (X,Y) parameters.
                mov      dl, X_gxy
                int      10h           ;Make the BIOS call.

                mov      sp, bp
                pop      bp
                ret
_GotoXY        endp


; ClrScrn-    Clears the screen and positions the cursor at (0,0).
;
; procedure ClrScrn;
; begin
;       BIOS(Initialize)
; end;
;
; Activation record for ClrScrn
;
;       |                        |
;       | Previous stk contents |
;       -------------------------
;       |  Return Adrs (offset) |
;       ------------------------- <- SP


                public  _ClrScrn
_ClrScrn        proc     near

                mov      ah, 6         ;Magic BIOS number.
                mov      al, 0         ;Clear entire screen.
                mov      bh, 07        ;Clear with black spaces.
                mov      cx, 0000      ;Upper left corner is (0,0)
                mov      dl, 79        ;Lower X-coordinate
                mov      dh, 24        ;Lower Y-coordinate
                int      10h           ;Make the BIOS call.

                push     0             ;Position the cursor to (0,0)
                push     0             ; after the call.
                call     _GotoXY
                add      sp, 4         ;Remove parameters from stack.

                ret
_ClrScrn        endp
```

```
; tstKbd-       Checks to see if a key is available at the keyboard.
;
; function tstKbd:boolean;
; begin
;       if BIOSKeyAvail then eat key and return true
;       else return false;
; end;
;
; Activation record for tstKbd
;
;       |                        |
;       | Previous stk contents |
;       -------------------------
;       |  Return Adrs (offset) |
;       ------------------------- <- SP

                public   _tstKbd
_tstKbd         proc     near
                mov      ah, 1          ;Check to see if key avail.
                int      16h
                je       NoKey
                mov      ah, 0          ;Eat the key if there is one.
                int      16h
                mov      ax, 1          ;Return true.
                ret

NoKey:          mov      ax, 0          ;No key, so return false.
                ret
_tstKbd         endp


; GetXY- Returns the cursor's current X and Y coordinates.
;
; procedure GetXY(var x:integer; var y:integer);
;
; Activation record for GetXY
;
;       |                        |
;       | Previous stk contents |
;       -------------------------
;       |    Y Coordinate        |
;       ---    Address         ---
;       |                        |
;       -------------------------
;       |    X coordinate        |
;       ---    Address         ---
;       |                        |
;       -------------------------
;       |  Return Adrs (offset) |
;       -------------------------
;       |      Old BP            |
;       ------------------------- <- BP
;       | Registers, etc.        |
;       ------------------------- <- SP


GXY_X           textequ  <[bp+4]>
GXY_Y           textequ  <[bp+8]>

                public   _GetXY
_GetXY          proc     near
                push     bp
                mov      bp, sp
                push     es

                mov      ah, 3          ;Read X, Y coordinates from
                mov      bh, 0          ; BIOS
                int      10h
```

```
                les     bx, GXY_X
                mov     es:[bx], dl
                mov     byte ptr es:[bx+1], 0

                les     bx, GXY_Y
                mov     es:[bx], dh
                mov     byte ptr es:[bx+1], 0

                pop     es
                pop     bp
                ret
_GetXY          endp


; PutChar- Outputs a single character to the screen at the current
;          cursor position.
;
; procedure PutChar(ch:char);
;
; Activation record for PutChar
;
;       |                        |
;       | Previous stk contents |
;       -------------------------
;       | char (in L.O. byte    |
;       -------------------------
;       |  Return Adrs (offset) |
;       -------------------------
;       |      Old BP           |
;       ------------------------- <- BP
;       | Registers, etc.       |
;       ------------------------- <- SP


ch_pc           textequ  <[bp+4]>

                public  _PutChar
_PutChar        proc    near
                push    bp
                mov     bp, sp

                mov     al, ch_pc
                mov     ah, 0eh
                int     10h

                pop     bp
                ret
_PutChar        endp



; PutStr-  Outputs a string to the display at the current cursor position.
;          Note that a string is a sequence of characters that ends with
;          a zero byte.
;
; procedure PutStr(var str:string);
;
; Activation record for PutStr
;
```

```
;       |                        |
;       | Previous stk contents |
;       -------------------------
;       |      String           |
;       ---    Address        ---
;       |                        |
;       -------------------------
;       |  Return Adrs (offset) |
;       -------------------------
;       |      Old BP           |
;       ------------------------- <- BP
;       | Registers, etc.       |
;       ------------------------- <- SP


Str_ps          textequ  <[bp+4]>

                public   _PutStr
_PutStr         proc     near
                push     bp
                mov      bp, sp
                push     es

                les      bx, Str_ps
PS_Loop:        mov      al, es:[bx]
                cmp      al, 0
                je       PC_Done

                push     ax
                call     _PutChar
                pop      ax
                inc      bx
                jmp      PS_Loop

PC_Done:        pop      es
                pop      bp
                ret
_PutStr         endp



; StartPgm-     This is where DOS starts running the program.  This is
;               a substitute for the C0L.OBJ file normally linked in by
;               the Borland C++ compiler.  This code provides this
;               routine to avoid legal problems (i.e., distributing
;               unlinked Borland libraries).  You can safely ignore
;               this code.  Note that the C++ main program is a near
;               procedure, so this code needs to be in the _TEXT segment.

                extern   _main:near
StartPgm        proc     near

                mov      ax, _DATA
                mov      ds, ax
                mov      es, ax
                mov      ss, ax
                lea      sp, EndStk

                call     near ptr _main
                mov      ah, 4ch
                int      21h
StartPgm        endp

_TEXT           ends

_DATA           segment  word public "DATA"
stack           word     1000h dup (?)
EndStk          word     ?
_DATA           ends
```

```
sseg            segment  para stack 'STACK'
                word     1000h dup (?)
sseg            ends
                end      StartPgm
```

11.12 Programming Projects

1) Write a version of the matrix multiply program inputs two 4x4 integer matrices from the user and compute their matrix product (see Chapter Eight question set). The matrix multiply algorithm (computing C := A * B) is

```
for i := 0 to 3 do
          for j := 0 to 3 do begin

                   c[i,j] := 0;
                   for k := 0 to 3 do
                       c[i,j] := c[i,j] + a[i,k] * b[k,j];
          end;
```

The program should have three procedures: InputMatrix, PrintMatrix, and MatrixMul. They have the following prototypes:

```
Procedure InputMatrix(var m:matrix);
procedure PrintMatrix(var m:matrix);
procedure MatrixMul(var result, A, B:matrix);
```

In particular note that these routines all pass their parameters by reference. Pass these parameters by reference on the stack.

Maintain all variables (e.g., i, j, k, etc.) on the stack using the techniques outlined in “Local Variable Storage” on page 604. In particular, do not keep the loop control variables in register.

Write a main program that makes appropriate calls to these routines to test them.

2) A pass by lazy evaluation parameter is generally a structure with three fields: a pointer to the thunk to call to the function that computes the value, a field to hold the value of the parameter, and a boolean field that contains false if the value field is uninitialized (the value field becomes initialized if the procedure writes to the value field or calls the thunk to obtain the value). Whenever the procedure writes a value to a pass by lazy evaluation parameter, it stores the value in the value field and sets the boolean field to true. Whenever a procedure wants to read the value, it first checks this boolean field. If it contains a true value, it simply reads the value from the value field; if the boolean field contains false, the procedure calls the thunk to compute the initial value. On return, the procedure stores the thunk result in the value field and sets the boolean field to true. Note that during any single activation of a procedure, the thunk for a parameter will be called, at most, one time. Consider the following Panacea procedure:

```
SampleEval: procedure(select:boolean; eval a:integer; eval b:integer);
var
       result:integer;
endvar;
begin SimpleEval;

       if (select) then

                 result := a;

       else
                 result := b;

       endif;
       writeln(result+2);

end SampleEval;
```

Write an assembly language program that implements SampleEval. From your main program call SampleEval a couple of times passing it different thunks for the a and b parameters. Your thunks can simply return a single value when called.

3) Write a shuffle routine to which you pass an array of 52 integers by reference. The routine should fill the array with the values 1..52 and then randomly shuffle the items in the array. Use the Standard Library random and randomize routines to select an index in the array to swap. See Chapter Seven, "Random Number Generation: Random, Randomize" on page 343 for more details about the random function. Write a main program that passes an array to this procedure and prints out the result.

11.13 Summary

In an assembly language program, all you need is a call and ret instruction to implement procedures and functions. Chapter Seven covers the basic use of procedures in an 80x86 assembly language program; this chapter describes how to organize program units like procedures and functions, how to pass parameters, allocate and access local variables, and related topics.

This chapter begins with a review of what a procedure is, how to implement procedures with MASM, and the difference between near and far procedures on the 80x86. For details, see the following sections:

- "Procedures" on page 566
- "Near and Far Procedures" on page 568
- "Forcing NEAR or FAR CALLs and Returns" on page 568
- "Nested Procedures" on page 569

Functions are a very important construct in high level languages like Pascal. However, there really isn't a difference between a function and a procedure in an assembly language program. Logically, a function returns a result and a procedure does not; but you declare and call procedures and functions identically in an assembly language program. See

- "Functions" on page 572

Procedures and functions often produce *side effects*. That is, they modify the values of registers and non-local variables. Often, these side effects are undesirable. For example, a procedure may modify a register that the caller needs preserved. There are two basic mechanisms for preserving such values: callee preservation and caller preservation. For details on these preservation schemes and other important issues see

- "Saving the State of the Machine" on page 572
- "Side Effects" on page 602

One of the major benefits to using a procedural language like Pascal or C++ is that you can easily pass parameters to and from procedures and functions. Although it is a little more work, you can pass parameters to your assembly language functions and procedures as well. This chapter discusses how and where to pass parameters. It also discusses how to access the parameters inside a procedure or function. To read about this, see sections

- "Parameters" on page 574
- "Pass by Value" on page 574
- "Pass by Reference" on page 575
- "Pass by Value-Returned" on page 575
- "Pass by Name" on page 576
- "Pass by Lazy-Evaluation" on page 577
- "Passing Parameters in Registers" on page 578
- "Passing Parameters in Global Variables" on page 580
- "Passing Parameters on the Stack" on page 581
- "Passing Parameters in the Code Stream" on page 590
- "Passing Parameters via a Parameter Block" on page 598

Since assembly language doesn't really support the notion of a function, per se, implementing a function consists of writing a procedure with a return parameter. As such, function results are quite similar to parameters in many respects. To see the similarities, check out the following sections:

- "Function Results" on page 600
- "Returning Function Results in a Register" on page 601
- "Returning Function Results on the Stack" on page 601
- "Returning Function Results in Memory Locations" on page 602

Most high level languages provide *local variable storage* associated with the activation and deactivation of a procedure or function. Although few assembly language programs use local variables in an identical fashion, it's very easy to implement dynamic allocation of local variables on the stack. For details, see section

- "Local Variable Storage" on page 604

Recursion is another HLL facility that is very easy to implement in an assembly language program. This chapter discusses the technique of recursion and then presents a simple example using the Quicksort algorithm. See

- "Recursion" on page 606

11.14 Questions

1) Explain how the CALL and RET instructions operate.

2) What are the operands for the PROC assembler directive? What is their function?

3) Rewrite the following code using PROC and ENDP:

```
        FillMem:            moval, 0FFh
FillLoop:                   mov[bx], al
                            incbx
                            loop FillLoop
                            ret
```

4) Modify your answer to problem (3) so that all affected registers are preserved by the FillMem procedure.

5) What happens if you fail to put a transfer of control instruction (such as a JMP or RET) immediately before the ENDP directive in a procedure?

6) How does the assembler determine if a CALL is near or far? How does it determine if a RET instruction is near or far?

7) How can you override the assembler's default decision whether to use a near or far CALL or RET?

8) Is there ever a need for nested procedures in an assembly language program? If so, give an example.

9) Give an example of why you might want to nest a segment inside a procedure.

10) What is the difference between a function, and a procedure?

11) Why should subroutines preserve the registers that they modify?

12) What are the advantages and disadvantages of caller-preserved values and callee-preserved values?

13) What are parameters?

14) How do the following parameter passing mechanisms work?

    a) Pass by value

    b) Pass by reference

    c) Pass by value-returned

    d) Pass by name

15) Where is the best place to pass parameters to a procedure?

16) List five different locations / methods for passing parameters to or from a procedure.

17) How are parameters that are passed on the stack accessed within a procedure?

18) What's the best way to deallocate parameters passed on the stack when the procedure terminates execution?

19) Given the following Pascal procedure definition:

```
procedure PascalProc(i,j,k:integer);
```

Explain how you would access the parameters of an equivalent assembly language program, assuming that the procedure is a near procedure.

20) Repeat problem (19) assuming that the procedure is a far procedure.

21) What does the stack look like during the execution of the procedure in problem (19)? Problem (20)?

22) How does an assembly language procedure gain access to parameters passed in the code stream?

23) How does the 80x86 skip over parameters passed in the code stream and continue program execution beyond them when the procedure returns to the caller?

24) What is the advantage to passing parameters via a parameter block?

25) Where are function results typically returned?

26) What is a side effect?

27) Where are local (temporary) variables typically allocated?

28) How do you allocate local (temporary) variables within a procedure?

29) Assuming you have three parameters passed by value on the stack and 4 different local variables, what does the activation record look like after the local variables have been allocated (assume a near procedure and no registers other than BP have been pushed onto the stack).

30) What is recursion?