**Chapter Seven**

**Detail of Procedures**

1. **Detail of Procedures**

*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 chapter will stick to simulating procedural constructs in 80x86 assembly language.

* 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. 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. 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. Without the ret instruction at the end of each procedure, the 80x86 will fall into the next subroutine rather than return to the caller.

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*.

* 1. **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:

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 empty3. 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.

* 1. **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. From here on, procedure will mean procedure or function.

* 1. **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, (very common)**

**• 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, (very common)**

• 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.

**Note:** from the above list, only those in bold are going to be discussed.

* + 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 C++ 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 Procedures and Functions strings by value is very inefficient (since you must create and pass a copy of the structure to the procedure).

* + 1. **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.

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.

* + 1. **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.

* + 1. **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

.

.

.

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. Fortunately, there are better parameters passing schemes for passing data in memory so you do not need to seriously consider this scheme.

* + 1. **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 CPP 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 7.1 (for a near procedure) or Figure 7.2 (for a far procedure). You could gain access to the parameters passed on the stack by removing the data rom the stack (Assuming a near procedure call):

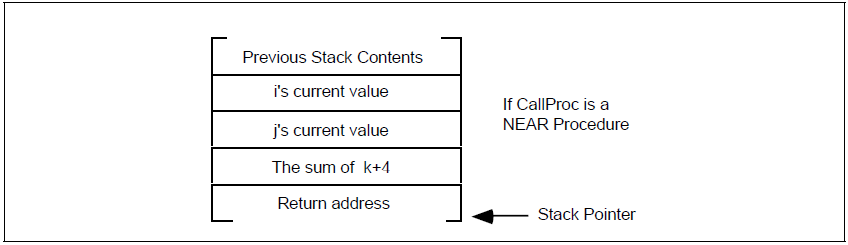


Figure 7.1 CallProc Stack Layout for a Near Procedure

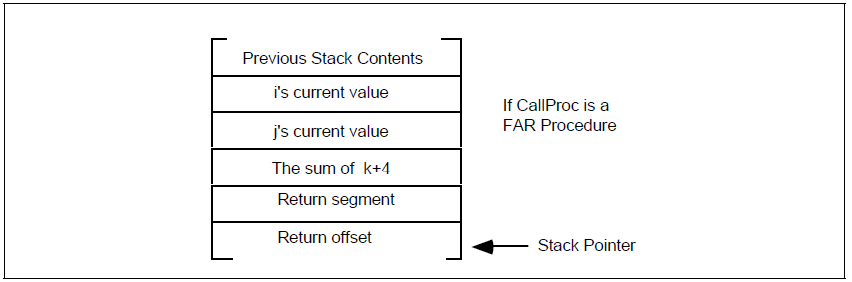


Figure 7.2 CallProc Stack Layouts for a Far Procedure

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 7.3.

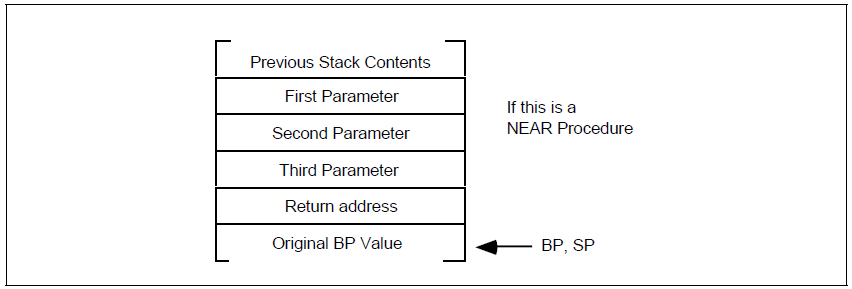


Figure 7.3 Accessing Parameters on the Stack

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 7.4.

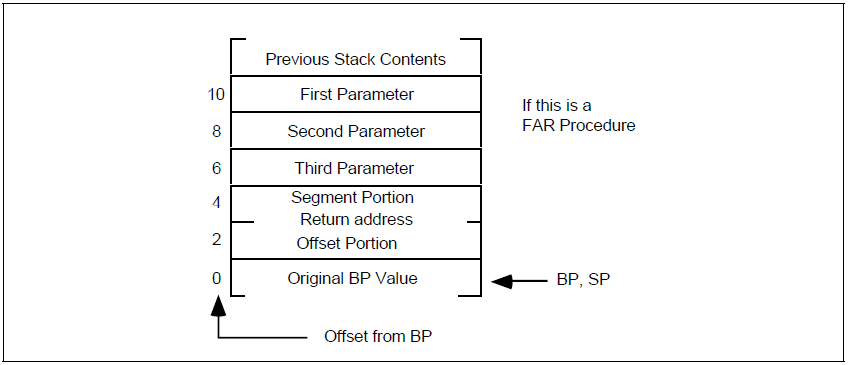


Figure 7.4 Accessing Parameters on the Stack in a Far Procedure

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 7.5.

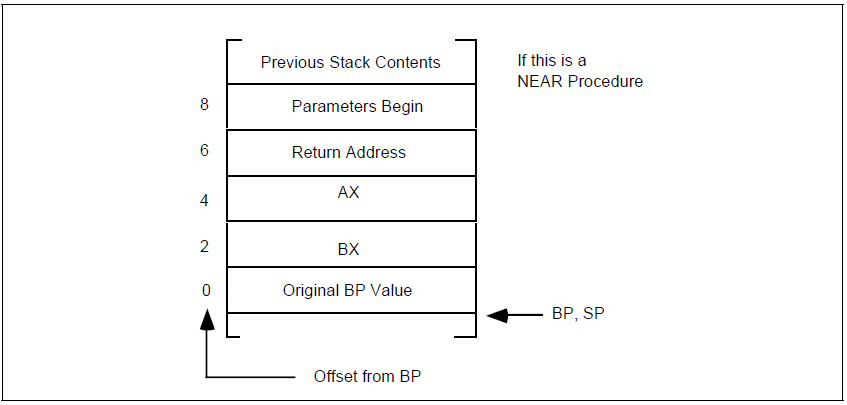


Figure 7.5 Messing up Offsets by Pushing Other Registers before BP

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 7.6).

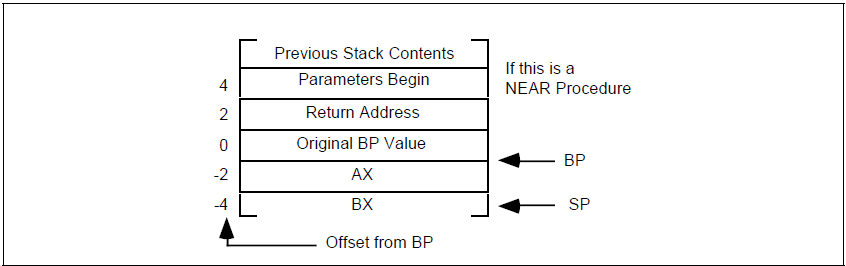


Figure 7.6 Keeping the Offsets Constant by Pushing BP First

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.

* 1. **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. Assembly language doesn’t distinguish between them.

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

* + 1. **Returning Function Results in a Register**

Like parameters, the 80x86’s registers are the best place to return function results. 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.

* 1. **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.

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:

LocalStuff proc near

push bp

mov bp, sp

push ax

sub sp, 6 ;Allocate local variables.

L0: mov ax, 8[bp]

add ax, 2

mov -4[bp], ax

mov ax, -4[bp]

mul 4[bp]

add ax, 6[bp]

mov 6[bp], ax

sub ax, -4[bp];AX already contains j

mov -8[bp], ax

add ax, -4[bp];AX already contains n

add ax, 6[bp]

mov -6[bp], 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 7.7.

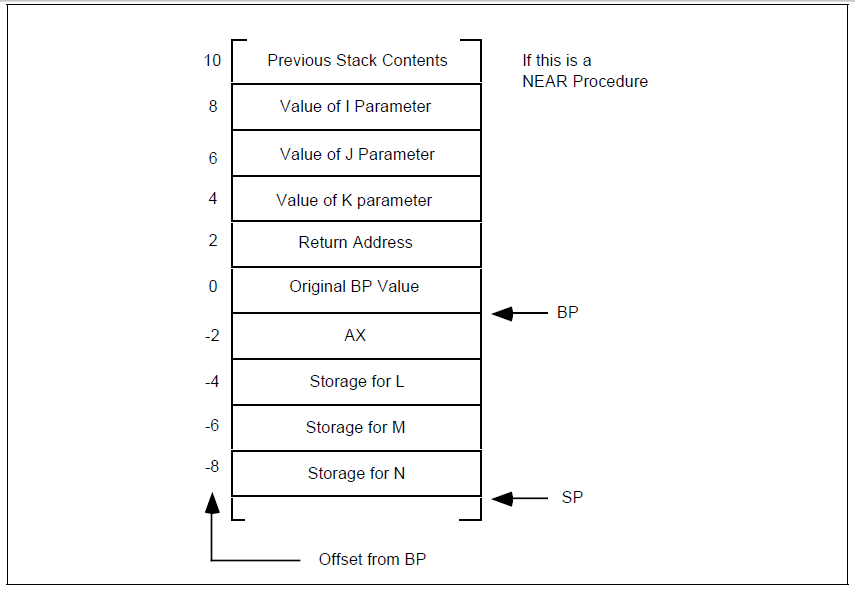


Figure 7.7 the Stack upon Entering the Next Procedure

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.