C H A P T E R 3

Using Addresses and Pointers

MASM applications running in real mode require segmented addresses to access code and data. The address of the code or data in a segment is relative to a segment address in a segment register. You can also use pointers to access data in assembly language programs. (A pointer is a variable that contains an address as its value.)

The first section of this chapter describes how to initialize default segment registers to access near and far addresses. The next section describes how to access code and data. It also describes related operators, syntax, and displacements. The discussion of memory operands lays the foundation for the third section, which describes the stack.

The fourth section of this chapter explains how to use the **TYPEDEF** directive to declare pointers and the **ASSUME** directive to give the assembler information about registers containing pointers. This section also shows you how to do typical pointer operations and how to write code that works for pointer variables in any memory model.

Programming Segmented Addresses

Before you use segmented addresses in your programs, you need to initialize the segment registers. The initialization process depends on the registers used and on your choice of simplified segment directives or full segment definitions. The simplified segment directives (introduced in Chapter 2) handle most of the initialization process for you. This section explains how to inform the assembler and the processor of segment addresses, and how to access the near and far code and data in those segments.

Initializing Default Segment Registers

The segmented architecture of the 8086-family of processors does not require that you specify two addresses every time you access memory. As explained in Chapter 2, “Organizing Segments,” the 8086 family of processors uses a system

of default segment registers to simplify access to the most commonly used data and code.

The segment registers DS, SS, and CS are normally initialized to default segments at the beginning of a program. If you write the main module in a high-level language, the compiler initializes the segment registers. If you write the main module in assembly language, you must initialize the segment registers yourself. Follow these steps to initialize segments:

1. Tell the assembler which segment is associated with a register. The assembler must know the default segments at assembly time.
2. Tell the processor which segment is associated with a register by writing the necessary code to load the correct segment value into the segment register on the processor.

These steps are discussed separately in the following sections.

Informing the Assembler About Segment Values

The first step in initializing segments is to tell the assembler which segment to associate with a register. You do this with the **ASSUME** directive. If you use simplified segment directives, the assembler automatically generates the appropriate **ASSUME** statements. If you use full segment definitions, you must code the **ASSUME** statements for registers other than CS yourself. (**ASSUME** can also be used on general-purpose registers, as explained in “Defining Register Types with ASSUME” later in this chapter.)

The **.STARTUP** directive generates startup code that sets DS equal to SS (unless you specify **FARSTACK**), allowing default data to be accessed through either SS or DS. This can improve efficiency in the code generated by compilers. The “DS equals SS” convention may not work with certain applications, such as memory-resident programs in MS-DOS and Windows dynamic-link libraries (see Chapter 10). The code generated for **.STARTUP** is shown in “Starting and Ending Code with .STARTUP and .EXIT” in Chapter 2. You can use similar code to set DS equal to SS in programs using full segment definitions.

Here is an example of **ASSUME** using full segment definitions:

```
ASSUME cs:_TEXT, ds:DGROUP, ss:DGROUP
```

This example is equivalent to the **ASSUME** statement generated with simplified segment directives in small model with **NEARSTACK**. Note that DS and SS are part of the same segment group. It is also possible to have different segments for data and code, and to use **ASSUME** to set ES, as shown here:

```
ASSUME cs:MYCODE, ds:MYDATA, ss:MYSTACK, es:OTHER
```

Correct use of the **ASSUME** statement can help find addressing errors. With **.CODE**, the assembler assumes CS is the current segment. When you use the simplified segment directives **.DATA**, **.DATA?**, **.CONST**, **.FARDATA**, or **.FARDATA?**, the assembler automatically assumes CS is the **ERROR** segment. This prevents instructions from appearing in these segments. If you use full segment definitions, you can accomplish the same by placing **ASSUME CS:ERROR** in a data segment.

With simple or full segments, you can cancel the control of an **ASSUME** statement by assuming **NOTHING**. You can cancel the previous assumption for ES with the following statement:

```
ASSUME es:NOTHING
```

Prior to the **.MODEL** statement (or in its absence), the assembler sets the **ASSUME** statement for DS, ES, and SS to the current segment.

Informing the Processor About Segment Values

The second and final step in initializing segments is to inform the processor of segment values at run time. How segment values are initialized at run time differs for each segment register and depends on the operating system and on your use of simplified segment directives or full segment definitions.

Specifying a Starting Address

A program's starting address determines where execution begins. After the operating system loads a program, it simply jumps to the starting address, giving processor control to the program. The true starting address is known only to the loader; the linker determines only the offset of the address within an undetermined code segment. That's why a normal application is often referred to as "relocatable code," because it runs regardless of where the loader places it in memory.

The offset of the starting address depends on the program type. Programs with an .EXE extension contain a header from which the loader reads the offset and combines it with a segment to form the starting address. Programs with a .COM extension (tiny model) have no such header, so by convention the loader jumps to the first byte of the program.

In either case, the **.STARTUP** directive identifies where execution begins, provided you use simplified segment directives. For an .EXE program, place **.STARTUP** immediately before the instruction where you want execution to start. In a .COM program, place **.STARTUP** before the first assembly instruction in your source code.

If you use full segment directives or prefer not to use **.STARTUP**, you must identify the starting instruction in two steps:

1. Label the starting instruction.
2. Provide the same label in the **END** directive.

These steps tell the linker where execution begins in the program. The following example illustrates the two steps for a tiny model program:

```
_TEXT   SEGMENT WORD PUBLIC 'CODE'
        ORG     100h    ; Use this declaration for .COM files only
start:  .               ; First instruction here
        .
        .
_TEXT   ENDS
        END     start   ; Name of starting label
```

Notice the **ORG** statement in this example. This statement is mandatory in a tiny model program without the **.STARTUP** directive. It places the first instruction at offset 100h in the code segment to create space for a 256-byte (100h) data area called the Program Segment Prefix (PSP). The operating system takes care of initializing the PSP, so you need only make sure the area exists. (For a description of what data resides in the PSP, refer to the "Tables" chapter in the *Reference*.)

Initializing DS

The DS register is automatically initialized to the correct value (DGROUP) if you use **.STARTUP** or if you are writing a program for Windows. If you do not use **.STARTUP** with MS-DOS, you must initialize DS using the following instructions:

```
        mov     ax, DGROUP
        mov     ds, ax
```

The initialization requires two instructions because the segment name is a constant and the assembler does not allow a constant to be loaded directly to a segment register. The previous example loads DGROUP, but you can load any valid segment or group.

Initializing SS and SP

The SS and SP registers are initialized automatically if you use the **.STACK** directive with simplified segments or if you define a segment that has the **STACK** combine type with full segment definitions. Using the **STACK** directive initializes SS to the stack segment. If you want SS to be equal to DS, use **.STARTUP** or its equivalent. (See "Combining Segments," page 45.) For an .EXE file, the stack address is encoded into the executable header and resolved

at load time. For a .COM file, the loader sets SS equal to CS and initializes SP to 0FFFEh.

If your program does not access far data, you do not need to initialize the ES register. If you choose to initialize, use the same technique as for the DS register. You can initialize SS to a far stack in the same way.

Near and Far Addresses

Addresses that have an implied segment name or segment registers associated with them are called "near addresses." Addresses that have an explicit segment associated with them are called "far addresses." The assembler handles near and far code automatically, as described in the following sections. You must specify how to handle far data.

The Microsoft segment model puts all near data and the stack in a group called DGROUP. Near code is put in a segment called _TEXT. Each module's far code or far data is placed in a separate segment. This convention is described in "Controlling the Segment Order" in Chapter 2.

The assembler cannot determine the address for some program components; these are said to be relocatable. The assembler generates a fixup record and the linker provides the address once it has determined the location of all segments. Usually a relocatable operand references a label, but there are exceptions. Examples in the next two sections include information about relocating near and far data.

Near Code

Control transfers within near code do not require changes to segment registers. The processor automatically handles changes to the offset in the IP register when control-flow instructions such as **JMP**, **CALL**, and **RET** are used. The statement

```
        call    nearproc        ; Change code offset
```

changes the IP register to the new address but leaves the segment unchanged. When the procedure returns, the processor resets IP to the offset of the next instruction after the **CALL** instruction.

Far Code

The processor automatically handles segment register changes when dealing with far code. The statement

```
        call    farproc         ; Change code segment and offset
```

automatically moves the segment and offset of the `farproc` procedure to the CS and IP registers. When the procedure returns, the processor sets CS to the

original code segment and sets IP to the offset of the next instruction after the call.

Near Data

A program can access near data directly, because a segment register already holds the correct segment for the data item. The term "near data" is often used to refer to the data in the DGROUP group.

After the first initialization of the DS and SS registers, these registers normally point into DGROUP. If you modify the contents of either of these registers during the execution of the program, you must reload the register with DGROUP's address before referencing any DGROUP data.

The processor assumes all memory references are relative to the segment in the DS register, with the exception of references using BP or SP. The processor associates these registers with the SS register. (You can override these assumptions with the segment override operator, described in "Direct Memory Operands," on page 62.)

The following lines illustrate how the processor accesses either the DS or SS segments, depending on whether the pointer operand contains BP or SP. Note the distinction loses significance when DS and SS are equal.

```
nearvar WORD    0
        .
        .
        .
        mov     ax, nearvar ; Reads from DS:[nearvar]
        mov     di, [bx]    ; Reads from DS:[bx]
        mov     [di], cx    ; Writes to  DS:[di]
        mov     [bp+6], ax  ; Writes to  SS:[bp+6]
        mov     bx, [bp]    ; Reads from SS:[bp]
```

Far Data

To read or modify a far address, a segment register must point to the segment of the data. This requires two steps. First load the segment (normally either ES or DS) with the correct value, and then (optionally) set an assume of the segment register to the segment of the address.

Note Flat model does not require far addresses. By default, all addressing is relative to the initial values of the segment registers. Therefore, this section on far addressing does not apply to flat model programs.

One method commonly used to access far data is to initialize the ES segment register. This example shows two ways to do this:

```
; First method
        mov     ax, SEG farvar  ; Load segment of the
        mov     es, ax          ,   far address into ES
        mov     ax, es:farvar   ; Provide an explicit segment
                                ;   override on the addressing
```

```
; Second method
        mov     ax, SEG farvar2 ; Load the segment of the
        mov     es, ax          ;   far address into ES
        ASSUME  ES:SEG farvar2  ; Tell the assembler that ES points
                                ;   to the segment containing farvar2
        mov     ax, farvar2     ; The assembler provides the ES
                                ;   override since it knows that
                                ;   the label is addressable
```

After loading the segment of the address into the ES segment register, you can explicitly override the segment register so that the addressing is correct (method 1) or allow the assembler to insert the override for you (method 2). The assembler uses **ASSUME** statements to determine which segment register can be used to address a segment of memory. To use the segment override operator, the left operand must be a segment register, not a segment name. (For more information on segment overrides, see "Direct Memory Operands" on page 62.)

If an instruction needs a segment override, the resulting code is slightly larger and slower, since the override must be encoded into the instruction. However, the resulting code may still be smaller than the code for multiple loads of the default segment register for the instruction.

The DS, SS, FS, and GS segment registers (FS and GS are available only on the 80386/486 processors) may also be used for addressing through other segments.

If a program uses ES to access far data, it need not restore ES when finished (unless the program uses flat model). However, some compilers require that you restore ES before returning to a module written in a high-level language.

To access far data, first set DS to the far segment and then restore the original DS when finished. Use the **ASSUME** directive to let the assembler know that DS no longer points to the default data segment, as shown here:

```
        push    ds               ; Save original segment
        mov     ax, SEG fararray ; Move segment into data register
        mov     ds, ax           ; Initialize segment register
        ASSUME  ds:SEG fararray  ; Tell assembler where data is
        mov     ax, fararray[0]  ; Set DX:AX = dword variable
        mov     dx, fararray[2]  ;   fararray
        .
        .
        .
        pop     ds               ; Restore segment
        ASSUME  ds:@DATA         ;   and default assumption
```

"Direct Memory Operands,"on page 62, describes an alternative method for accessing far data. The technique of resetting DS as shown in the previous example is best for a lengthy series of far data references. The segment override method described in "Direct Memory Operands" serves best when accessing only one or two far variables.

If your program changes DS to access far data, it should restore DS when finished. This allows procedures to assume that DS is the segment for near data. Many compilers, including Microsoft compilers, use this convention.

Operands

With few exceptions, assembly language instructions work on sources of data called operands. In a listing of assembly code (such as the examples in this book), operands appear in the operand field immediately to the right of the instructions.

This section describes the four kinds of instruction operands: register, immediate, direct memory, and indirect memory. Some instructions, such as POPF and STI, have implied operands which do not appear in the operand field. Otherwise, an implied operand is just as real as one stated explicitly.

Certain other instructions such as NOP and WAIT deserve special mention. These instructions affect only processor control and do not require an operand.

The following four types of operands are described in the rest of this section:

Operand Type	Addressing Mode
Register	An 8-bit or 16-bit register on the 8086–80486; can also be 32-bit on the 80386/486.
Immediate	A constant value contained in the instruction itself.
Direct memory	A fixed location in memory.
Indirect memory	A memory location determined at run time by using the address stored in one or two registers.

Instructions that take two or more operands always work right to left. The right operand is the source operand. It specifies data that will be read, but not changed, in the operation. The left operand is the destination operand. It specifies the data that will be acted on and possibly changed by the instruction.

Register Operands

Register operands refer to data stored in registers. The following examples show typical register operands:

```
mov     bx, 10          ; Load constant to BX
add     ax, bx          ; Add BX to AX
jmp     di              ; Jump to the address in DI
```

An offset stored in a base or index register often serves as a pointer into memory. You can store an offset in one of the base or index registers, then use the register as an indirect memory operand. (See "Indirect Memory Operands," following.) For example:

```
mov     [bx], dl ; Store DL in indirect memory operand
inc     bx       ; Increment register operand
mov     [bx], dl ; Store DL in new indirect memory operand
```

This example moves the value in DL to 2 consecutive bytes of a memory location pointed to by BX. Any instruction that changes the register value also changes the data item pointed to by the register.

Immediate Operands

An immediate operand is a constant or the result of a constant expression. The assembler encodes immediate values into the instruction at assembly time. Here are some typical examples showing immediate operands:

```
mov     cx, 20          ; Load constant to register
add     var, 1Fh        ; Add hex constant to variable
sub     bx, 25 * 80     ; Subtract constant expression
```

Immediate data is never permitted in the destination operand. If the source operand is immediate, the destination operand must be either a register or direct memory to provide a place to store the result of the operation.

Immediate expressions often involve the useful **OFFSET** and **SEG** operators, described in the following paragraphs.

The OFFSET Operator

An address constant is a special type of immediate operand that consists of an offset or segment value. The **OFFSET** operator returns the offset of a memory location, as shown here:

```
mov     bx, OFFSET var  ; Load offset address
```

For information on differences between MASM 5.1 behavior and MASM 6.1 behavior related to **OFFSET**, see Appendix A.

Since data in different modules may belong to a single segment, the assembler cannot know for each module the true offsets within a segment. Thus, the offset for **var**, although an immediate value, is not determined until link time.

The SEG Operator

The **SEG** operator returns the segment of a memory location:

```
mov     ax, SEG farvar  ; Load segment address
mov     es, ax
```

The actual value of a particular segment is not known until the program is loaded into memory. For .EXE programs, the linker makes a list in the program's header of all locations in which the **SEG** operator appears. The loader reads this list and fills in the required segment address at each location. Since .COM programs have no header, the assembler does not allow relocatable segment expressions in tiny model programs.

The **SEG** operator returns a variable's "frame" if it appears in the instruction. The frame is the value of the segment, group, or segment override of a nonexternal variable. For example, the instruction

```
mov     ax, SEG DGROUP:var
```

places in AX the value of DGROUP, where **var** is located. If you do not include a frame, **SEG** returns the value of the variable's group if one exists. If the variable is not defined in a group, **SEG** returns the variable's segment address.

This behavior can be changed with the /Zm command-line option or with the **OPTION OFFSET:SEGMENT** statement. (See Appendix A, "Differences between MASM 6.1 and 5.1.") "Using the OPTION Directive" in Chapter 1 introduces the **OPTION** directive.

Direct Memory Operands

A direct memory operand specifies the data at a given address. The instruction acts on the contents of the address, not the address itself. Except when size is implied by another operand, you must specify the size of a direct memory operand so the instruction accesses the correct amount of memory. The following example shows how to explicitly specify data size with the **BYTE** directive:

```
        .DATA?                  ; Segment for uninitialized data
var     BYTE   ?                ; Reserve one byte, labeled "var"
        .CODE
        .
        .
        .
        mov    var, al          ; Copy AL to byte at var
```

Any location in memory can be a direct memory operand as long as a size is specified (or implied) and the location is fixed. The data at the address can change, but the address cannot. By default, instructions that use direct memory addressing use the DS register. You can create an expression that points to a memory location using any of the following operators:

Operator Name	Symbol
Plus	+
Minus	–
Index	[ ]
Structure member	.
Segment override	:

These operators are discussed in more detail in the following section.

Plus, Minus, and Index

The plus and index operators perform in exactly the same way when applied to direct memory operands. For example, both the following statements move the second word value from an array into the AX register:

```
        mov     ax, array[2]
        mov     ax, array+2
```

The index operator can contain any direct memory operand. The following statements are equivalent:

```
        mov     ax, var
        mov     ax, [var]
```

Some programmers prefer to enclose the operand in brackets to show that the contents, not the address, are used.

The minus operator behaves as you would expect. Both the following instructions retrieve the value located at the word preceding **array**:

```
mov     ax, array[-2]
mov     ax, array-2
```

Structure Field

The structure operator (**.**) references a particular element of a structure or "field," to use C terminology:

```
mov     bx, structvar.field1
```

The address of the structure operand is the sum of the offsets of **structvar** and **field1**. For more information about structures, see "Structures and Unions" in Chapter 5.

Segment Override

The segment override operator (**:**) specifies a segment portion of the address that is different from the default segment. When used with instructions, this operator can apply to segment registers or segment names:

```
mov     ax, es:farvar           ; Use segment override
```

The assembler will not generate a segment override if the default segment is explicitly provided. Thus, the following two statements assemble in exactly the same way:

```
mov     [bx], ax
mov     ds:[bx], ax
```

A segment name override or the segment override operator identifies the operand as an address expression.

```
        mov     WORD PTR FARSEG:0, ax   ; Segment name override
        mov     WORD PTR es:100h, ax    ; Legal and equivalent
        mov     WORD PTR es:[100h], ax  ;   expressions
;       mov     WORD PTR [100h], ax     ; Illegal, not an address
```

As the example shows, a constant expression cannot be an address expression unless it has a segment override.

Indirect Memory Operands

Like direct memory operands, indirect memory operands specify the contents of a given address. However, the processor calculates the address at run time by referring to the contents of registers. Since values in the registers can change at run time, indirect memory operands provide dynamic access to memory.

Indirect memory operands make possible run-time operations such as pointer indirection and dynamic indexing of array elements, including indexing of multidimensional arrays.

Strict rules govern which registers you can use for indirect memory operands under 16-bit versions of the 8086-based processors. The rules change significantly for 32-bit processors starting with the 80386. However, the new rules apply only to code that does not need to be compatible with earlier processors.

This section covers features of indirect operands in either mode. The specific 16-bit rules and 32-bit rules are then explained separately.

Indirect Operands with 16- and 32-Bit Registers

Some rules and options for indirect memory operands always apply, regardless of the size of the register. For example, you must always specify the register and operand size for indirect memory operands. But you can use various syntaxes to indicate an indirect memory operand. This section describes the rules that apply to both 16-bit and 32-bit register modes.

Specifying Indirect Memory Operands

The index operator specifies the register or registers for indirect operands. The processor uses the data pointed to by the register. For example, the following instruction moves into AX the word value at the address in DS:BX.

```
mov     ax, WORD PTR [bx]
```

When you specify more than one register, the processor adds the contents of the two addresses together to determine the effective address (the address of the data to operate on):

```
mov     ax, [bx+si]
```

Specifying Displacements

You can specify an address displacement, which is a constant value added to the effective address. A direct memory specifier is the most common displacement:

```
mov     ax, table[si]
```

In this relocatable expression, the displacement **table** is the base address of an array; SI holds an index to an array element. The SI value is calculated at run time, often in a loop. The element loaded into AX depends on the value of SI at the time the instruction executes.

Each displacement can be an address or numeric constant. If there is more than one displacement, the assembler totals them at assembly time and encodes the total displacement. For example, in the statement

```
table   WORD    100 DUP (0)
        .
        .
        .
        mov     ax, table[bx][di]+6
```

both **table** and **6** are displacements. The assembler adds the value of **6** to **table** to get the total displacement. However, the statement

```
        mov ax, mem1[si] + mem2
```

is not legal, because it attempts to use a single command to join the contents of two different addresses.

Specifying Operand Size

You must give the size of an indirect memory operand in one of three ways:

- By the variable's declared size
- With the **PTR** operator
- Implied by the size of the other operand

The following lines illustrate all three methods. Assume the size of the **table** array is **WORD**, as declared earlier.

```
        mov     table[bx], 0       ; 2 bytes - from size of table
        mov     BYTE PTR table, 0  ; 1 byte  - specified by BYTE
        mov     ax, [bx]           ; 2 bytes - implied by AX
```

Syntax Options

The assembler allows a variety of syntaxes for indirect memory operands. However, all registers must be inside brackets. You can enclose each register in its own pair of brackets, or you can place the registers in the same pair of brackets separated by a plus operator (+). All the following variations are legal and assemble the same way:

```
        mov     ax, table[bx][di]
        mov     ax, table[di][bx]
        mov     ax, table[bx+di]
        mov     ax, [table+bx+di]
        mov     ax, [bx][di]+table
```

All of these statements move the value in **table** indexed by BX+DI into AX.

Scaling Indexes

The value of index registers pointing into arrays must often be adjusted for zero-based arrays and scaled according to the size of the array items. For a word array, the item number must be multiplied by two (shifted left by one place). When using 16-bit registers, you must scale with separate instructions, as shown here:

```
        mov     bx, 5           ; Get sixth element (adjust for 0)
        shl     bx, 1           ; Scale by two (word size)
        inc     wtable[bx]      ; Increment sixth element in table
```

When using 32-bit registers on the 80386/486 processor, you can include scaling in the operand, as described in "Indirect Memory Operands with 32-Bit Registers," following.

Accessing Structure Elements

The structure member operator can be used in indirect memory operands to access structure elements. In this example, the structure member operator loads the **year** field of the fourth element of the **students** array into AL:

```
STUDENT STRUCT
  grade WORD    ?
  name  BYTE    20 DUP (?)
  year  BYTE    ?
STUDENT ENDS

students        STUDENT  < >
        .
        .                               ; Assume array is initialized
        mov     bx, OFFSET students     ; Point to array of students
        mov     ax, 4                   ; Get fourth element
        mov     di, SIZE STUDENT        ; Get size of STUDENT
        mul     di                      ; Multiply size times
        mov     di, ax                  ;   elements to point DI
                                        ;   to current element
        mov     al, (STUDENT PTR[bx+di]).year
```

For more information on MASM structures, see "Structures and Unions" in Chapter 5.

Indirect Memory Operands with 16-Bit Registers

For 8086-based computers and MS-DOS, you must follow the strict indexing rules established for the 8086 processor. Only four registers are allowed—BP, BX, SI, and DI—those only in certain combinations.

BP and BX are base registers. SI and DI are index registers. You can use either a base or an index register by itself. But if you combine two registers, one must be a base and one an index. Here are legal and illegal forms:

```
        mov     ax, [bx+di]     ; Legal
        mov     ax, [bx+si]     ; Legal
        mov     ax, [bp+di]     ; Legal
        mov     ax, [bp+si]     ; Legal
;       mov     ax, [bx+bp]     ; Illegal - two base registers
;       mov     ax, [di+si]     ; Illegal - two index registers
```

Table 3.1 shows the register modes in which you can specify indirect memory operands.

Table 3.1 Indirect Addressing with 16-Bit Registers

Mode	Syntax	Effective Address
Register indirect	[BX] [BP] [DI] [SI]	Contents of register
Base or index	*displacement*[BX] *displacement*[BP] *displacement*[DI] *displacement*[SI]	Contents of register plus *displacement*
Base plus index	[BX][DI] [BP][DI] [BX][SI] [BP][SI]	Contents of base register plus contents of index register
Base plus index with displacement	*displacement*[BX][DI] *displacement*[BP][DI] *displacement*[BX][SI] *displacement*[BP][SI]	Sum of base register, index register, and *displacement*

Different combinations of registers and displacements have different timings, as shown in *Reference*.

Indirect Memory Operands with 32-Bit Registers

You can write instructions for the 80386/486 processor using either 16-bit or 32-bit segments. Indirect memory operands are different in each case.

In 16-bit real mode, the 80386/486 operates the same way as earlier 8086-based processors, with one difference: you can use 32-bit registers. If the 80386/486 processor is enabled (with the **.386** or **.486** directive), 32-bit general-purpose registers are available with either 16-bit or 32-bit segments. Thirty-two–bit

registers eliminate many of the limitations of 16-bit indirect memory operands. You can use 80386/486 features to make your MS-DOS programs run faster and more efficiently if you are willing to sacrifice compatibility with earlier processors.

In 32-bit mode, an offset address can be up to 4 gigabytes. (Segments are still represented in 16 bits.) This effectively eliminates size restrictions on each segment, since few programs need 4 gigabytes of memory. Windows NT uses 32-bit mode and flat model, which spans all segments. XENIX 386 uses 32-bit mode with multiple segments.

80386/486 Enhancements

On the 80386/486, the processor allows you to use any general-purpose 32-bit register as a base or index register, except ESP, which can be a base but not an index. However, you cannot combine 16-bit and 32-bit registers. Several examples are shown here:

```
add     edx, [eax]              ; Add double
mov     dl, [esp+10]            ; Copy byte from stack
dec     WORD PTR [edx][eax]     ; Decrement word
cmp     ax, array[ebx][ecx]     ; Compare word from array
jmp     FWORD PTR table[ecx]    ; Jump into pointer table
```

Scaling Factors

With 80386/486 registers, the index register can have a scaling factor of 1, 2, 4, or 8. Any register except ESP can be the index register and can have a scaling factor. To specify the scaling factor, use the multiplication operator (*) adjacent to the register.

You can use scaling to index into arrays with different sizes of elements. For example, the scaling factor is 1 for byte arrays (no scaling needed), 2 for word arrays, 4 for doubleword arrays, and 8 for quadword arrays. There is no performance penalty for using a scaling factor. Scaling is illustrated in the following examples:

```
mov     eax, darray[edx*4]     ; Load double of double array
mov     eax, [esi*8][edi]      ; Load double of quad array
mov     ax, wtbl[ecx+2][edx*2] ; Load word of word array
```

Scaling is also necessary on earlier processors, but it must be done with separate instructions before the indirect memory operand is used, as described in "Indirect Memory Operands with 16-Bit Registers," previous.

The default segment register is SS if the base register is EBP or ESP. However, if EBP is scaled, the processor treats it as an index register with a value relative to DS, not SS.

All other base registers are relative to DS. If two registers are used, only one can have a scaling factor. The register with the scaling factor is defined as the index register. The other register is defined as the base. If scaling is not used, the first register is the base. If only one register is used, it is considered the base for deciding the default segment unless it is scaled. The following examples illustrate how to determine the base register:

```
mov   eax, [edx][ebp*4] ; EDX base (not scaled - seg DS)
mov   eax, [edx*1][ebp] ; EBP base (not scaled - seg SS)
mov   eax, [edx][ebp]   ; EDX base (first - seg DS)
mov   eax, [ebp][edx]   ; EBP base (first - seg SS)
mov   eax, [ebp]        ; EBP base (only - seg SS)
mov   eax, [ebp*2]      ; EBP*2 index (seg DS)
```

Mixing 16-Bit and 32-Bit Registers

Assembly statements can mix 16-bit and 32-bit registers. For example, the following statement is legal for 16-bit and 32-bit segments:

```
        mov     eax, [bx]
```

This statement moves the 32-bit value pointed to by BX into the EAX register. Although BX is a 16-bit pointer, it can still point into a 32-bit segment.

However, the following statement is never legal, since you cannot use the CX register as a 16-bit pointer:

```
;       mov     eax, [cx]       ; illegal
```

Operands that mix 16-bit and 32-bit registers are also illegal:

```
;       mov     eax, [ebx+si]   ; illegal
```

The following statement is legal in either 16-bit or 32-bit mode:

```
        mov     bx, [eax]
```

This statement moves the 16-bit value pointed to by EAX into the BX register. This works in 32-bit mode. However, in 16-bit mode, moving a 32-bit pointer into a 16-bit segment is illegal. If EAX contains a 16-bit value (the top half of the 32-bit register is 0), the statement works. However, if the top half of the EAX register is not 0, the operand points into a part of the segment that doesn't exist, generating an error. If you use 32-bit registers as indexes in 16-bit mode, you must make sure that the index registers contain valid 16-bit addresses.

The Program Stack

The preceding discussion on memory operands lays the groundwork for understanding the important data area known as the "stack."

A stack is an area of memory for storing data temporarily. Unlike other segments that store data starting from low memory, the stack stores data starting from high memory. Data is always pushed onto, or "popped" from the top of the stack.

The stack gets its name from its similarity to the spring-loaded plate holders in cafeterias. You add and remove plates from only the top of the stack. To retrieve the third plate, you must remove—that is, "pop"—the first two plates. Stacks are often referred to as LIFO buffers, from their last-in-first-out operation.

A stack is an essential part of any nontrivial program. A program continually uses its stack to temporarily store return addresses, procedure arguments, memory data, flags, or registers.

The SP register serves as an indirect memory operand to the top of the stack. At first, the stack is an uninitialized segment of a finite size. As your program adds data to the stack, the stack grows downward from high memory to low memory. When you remove items from the stack, it shrinks upward from low to high memory.

Saving Operands on the Stack

The **PUSH** instruction stores a 2-byte operand on the stack. The **POP** instruction retrieves the most recent pushed value. When a value is pushed onto the stack, the assembler decreases the SP (Stack Pointer) register by 2. On 8086-based processors, the SP register always points to the top of the stack. The **PUSH** and **POP** instructions use the SP register to keep track of the current position.

When a value is popped off the stack, the assembler increases the SP register by 2. Since the stack always contains word values, the SP register changes in multiples of two. When a **PUSH** or **POP** instruction executes in a 32-bit code segment (one with **USE32** use type), the assembler transfers a 4-byte value, and ESP changes in multiples of four.

Note The 8086 and 8088 processors differ from later Intel processors in how they push and pop the SP register. If you give the statement **push sp** with the 8086 or 8088, the word pushed is the word in SP after the push operation.

Figure 3.1 illustrates how pushes and pops change the SP register.

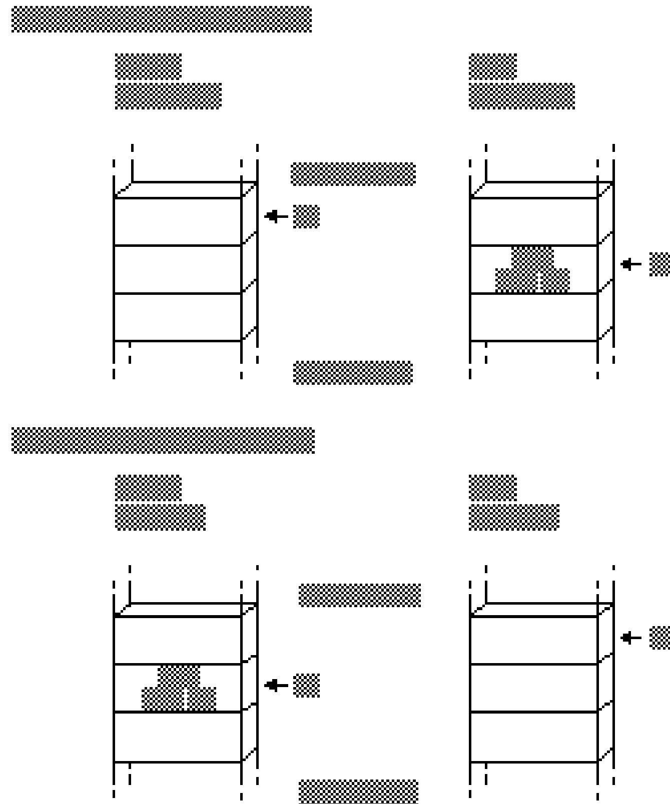

Figure 3.1 Stack Status Before and After Pushes and Pops

On the 8086, **PUSH** and **POP** take only registers or memory expressions as their operands. The other processors allow an immediate value to be an operand for **PUSH**. For example, the following statement is legal on the 80186–80486 processors:

```
push    7               ; 3 clocks on 80286
```

That statement is faster than these equivalent statements, which are required on the 8088 or 8086:

```
mov     ax, 7           ; 2 clocks plus
push    ax              ; 3 clocks on 80286
```

Words are popped off the stack in reverse order: the last item pushed is the first popped. To return the stack to its original status, you do the same number of

pops as pushes. You can subtract the correct number of words from the SP register if you want to restore the stack without using the values on it.

To reference operands on the stack, remember that the values pointed to by the BP (Base Pointer) and SP registers are relative to the SS (Stack Segment) register. The BP register is often used to point to the base of a frame of reference (a stack frame) within the stack. This example shows how you can access values on the stack using indirect memory operands with BP as the base register.

```
push    bp              ; Save current value of BP
mov     bp, sp          ; Set stack frame
push    ax              ; Push first;  SP = BP - 2
push    bx              ; Push second; SP = BP - 4
push    cx              ; Push third;  SP = BP - 6
.
.
.
mov     ax, [bp-6]      ; Put third word in AX
mov     bx, [bp-4]      ; Put second word in BX
mov     cx, [bp-2]      ; Put first word in CX
.
.
.
add     sp, 6           ; Restore stack pointer
                        ;   (two bytes per push)
pop     bp              ; Restore BP
```

If you often use these stack values in your program, you may want to give them labels. For example, you can use **TEXTEQU** to create a label such as **`count TEXTEQU <[bp-6]>`**. Now you can replace the **`mov ax, [bp - 6]`** statement in the previous example with **`mov ax, count`**. For more information about the **TEXTEQU** directive, see "Text Macros" in Chapter 9.

Saving Flags on the Stack

Your program can push and pop flags onto the stack with the **PUSHF** and **POPF** instructions. These instructions save and then restore the status of the flags. You can also use them within a procedure to save and restore the flag status of the caller. The 32-bit versions of these instructions are **PUSHFD** and **POPFD**.

This example saves the flags register before calling the **`systask`** procedure:

```
pushf
call    systask
popf
```

If you do not need to store the entire flags register, you can use the **LAHF** instruction to manually load and store the status of the lower byte of the flag register in the AH register. **SAHF** restores the value.

Saving Registers on the Stack (80186–80486 Only)

Starting with the 80186 processor, the **PUSHA** and **POPA** instructions push or pop all the general-purpose registers with only one instruction. These instructions save the status of all registers before a procedure call and restore them after the return. Using **PUSHA** and **POPA** is significantly faster and takes fewer bytes of code than pushing and popping each register individually.

The processor pushes the registers in the following order: AX, CX, DX, BX, SP, BP, SI, and DI. The SP word pushed is the value before the first register is pushed.

The processor pops the registers in the opposite order. The 32-bit versions of these instructions are **PUSHAD** and **POPAD**.

Accessing Data with Pointers and Addresses

A pointer is simply a variable that contains an address of some other variable. The address in the pointer "points" to the other object. Pointers are useful when transferring a large data object (such as an array) to a procedure. The caller places only the pointer on the stack, which the called procedure uses to locate the array. This eliminates the impractical step of having to pass the entire array back and forth through the stack.

There is a difference between a far address and a far pointer. A "far address" is the address of a variable located in a far data segment. A "far pointer" is a variable that contains the segment address and offset of some other data. Like any other variable, a pointer can be located in either the default (near) data segment or in a far segment.

Previous versions of MASM allow pointer variables but provide little support for them. In previous versions, any address loaded into a variable can be considered a pointer, as in the following statements:

```
Var     BYTE    0               ; Variable
npVar   WORD    Var             ; Near pointer to variable
fpVar   DWORD   Var             ; Far pointer to variable
```

If a variable is initialized with the name of another variable, the initialized variable is a pointer, as shown in this example. However, in previous versions of MASM, the CodeView debugger recognizes **npVar** and **fpVar** as word and doubleword variables. CodeView does not treat them as pointers, nor does it recognize the type of data they point to (bytes, in the example).

The **TYPEDEF** directive and enhanced capabilities of **ASSUME** (introduced in MASM 6.0) make it easier to manage pointers in registers and variables. The rest of this chapter describes these directives and how they apply to basic pointer operations.

Defining Pointer Types with TYPEDEF

The **TYPEDEF** directive can define types for pointer variables. A type so defined is considered the same as the intrinsic types provided by the assembler and can be used in the same contexts. When used to define pointers, the syntax for **TYPEDEF** is:

typename **TYPEDEF** ⟦*distance*⟧ **PTR** *qualifiedtype*

The *typename* is the name assigned to the new type. The *distance* can be **NEAR**, **FAR**, or any distance modifier. The *qualifiedtype* can be any previously intrinsic or defined MASM type, or a type previously defined with **TYPEDEF**. (For a full definition of *qualifiedtype*, see "Data Types" in Chapter 1.)

Here are some examples of user-defined types:

```
PBYTE   TYPEDEF      PTR BYTE   ; Pointer to bytes
NPBYTE  TYPEDEF NEAR PTR BYTE   ; Near pointer to bytes
FPBYTE  TYPEDEF FAR  PTR BYTE   ; Far pointer to bytes
PWORD   TYPEDEF      PTR WORD   ; Pointer to words
NPWORD  TYPEDEF NEAR PTR WORD   ; Near pointer to words
FPWORD  TYPEDEF FAR  PTR WORD   ; Far pointer to words

PPBYTE  TYPEDEF      PTR PBYTE  ; Pointer to pointer to bytes
                                ;   (in C, an array of strings)
PVOID   TYPEDEF      PTR        ; Pointer to any type of data

PERSON  STRUCT                  ; Structure type
  name  BYTE    20 DUP (?)
  num   WORD    ?
PERSON  ENDS
PPERSON TYPEDEF      PTR PERSON ; Pointer to structure type
```

The distance of a pointer can be set specifically or determined automatically by the memory model (set by **.MODEL**) and the segment size (16 or 32 bits). If you don't use **.MODEL**, near pointers are the default.

In 16-bit mode, a near pointer is 2 bytes that contain the offset of the object pointed to. A far pointer requires 4 bytes, and contains both the segment and offset. In 32-bit mode, a near pointer is 4 bytes and a far pointer is 6 bytes, since segments are

still word values in 32-bit mode. If you specify the distance with **NEAR** or **FAR**, the processor uses the default distance of the current segment size. You can use **NEAR16**, **NEAR32**, **FAR16**, and **FAR32** to override the defaults set by the current segment size. In flat model, **NEAR** is the default.

You can declare pointer variables with a pointer type created with **TYPEDEF**. Here are some examples using these pointer types.

```
; Type declarations
Array   WORD    25 DUP (0)
Msg     BYTE    "This is a string", 0
pMsg    PBYTE   Msg              ; Pointer to string
pArray  PWORD   Array            ; Pointer to word array
npMsg   NPBYTE  Msg              ; Near pointer to string
npArray NPWORD  Array            ; Near pointer to word array
fpArray FPWORD  Array            ; Far pointer to word array
fpMsg   FPBYTE  Msg              ; Far pointer to string

S1      BYTE    "first", 0       ; Some strings
S2      BYTE    "second", 0
S3      BYTE    "third", 0
pS123   PBYTE   S1, S2, S3, 0    ; Array of pointers to strings
ppS123  PPBYTE  pS123            ; A pointer to pointers to strings

Andy    PERSON  <>               ; Structure variable
pAndy   PPERSON Andy             ; Pointer to structure variable

                                 ; Procedure prototype

EXTERN  ptrArray:PBYTE           ; External variable
Sort    PROTO   pArray:PBYTE     ; Parameter for prototype

; Parameter for procedure
Sort    PROC    pArray:PBYTE
        LOCAL   pTmp:PBYTE       ; Local variable
        .
        .
        .
        ret
Sort    ENDP
```

Once defined, pointer types can be used in any context where intrinsic types are allowed.

Defining Register Types with ASSUME

You can use the **ASSUME** directive with general-purpose registers to specify that a register is a pointer to a certain size of object. For example:

```
ASSUME  bx:PTR WORD     ; Assume BX is now a word pointer
inc     [bx]            ; Increment word pointed to by BX
add     bx, 2           ; Point to next word
mov     [bx], 0         ; Word pointed to by BX = 0
.
.                       ; Other pointer operations with BX
.
ASSUME  bx:NOTHING      ; Cancel assumption
```

In this example, BX is specified as a pointer to a word. After a sequence of using BX as a pointer, the assumption is canceled by assuming **NOTHING**.

Without the assumption to **PTR WORD**, many instructions need a size specifier. The **INC** and **MOV** statements from the previous examples would have to be written like this to specify the sizes of the memory operands:

```
inc     WORD PTR [bx]
mov     WORD PTR [bx], 0
```

When you have used **ASSUME**, attempts to use the register for other purposes generate assembly errors. In this example, while the **PTR WORD** assumption is in effect, any use of BX inconsistent with its **ASSUME** declaration generates an error. For example,

```
;       mov     al, [bx]        ; Can't move word to byte register
```

You can also use the **PTR** operator to override defaults:

```
mov     al, BYTE PTR [bx]       ; Legal
```

Similarly, you can use **ASSUME** to prevent the use of a register as a pointer, or even to disable a register:

```
        ASSUME  bx:WORD, dx:ERROR
;       mov     al, [bx] ; Error - BX is an integer, not a pointer
;       mov     ax, dx   ; Error - DX disabled
```

For information on using **ASSUME** with segment registers, refer to "Setting the ASSUME Directive for Segment Registers" in Chapter 2.

Basic Pointer and Address Operations

A program can perform the following basic operations with pointers and addresses:

- Initialize a pointer variable by storing an address in it.
- Load an address into registers, directly or from a pointer.

The sections in the rest of this chapter describe variations of these tasks with pointers and addresses. The examples are used with the assumption that you have previously defined the following pointer types with the **TYPEDEF** directive:

```
PBYTE   TYPEDEF      PTR BYTE   ; Pointer to bytes
NPBYTE  TYPEDEF NEAR PTR BYTE   ; Near pointer to bytes
FPBYTE  TYPEDEF FAR  PTR BYTE   ; Far pointer to bytes
```

Initializing Pointer Variables

If the value of a pointer is known at assembly time, the assembler can initialize it automatically so that no processing time is wasted on the task at run time. The following example shows how to do this, placing the address of **msg** in the pointer **pmsg**.

```
Msg     BYTE    "String", 0
pMsg    PBYTE   Msg
```

If a pointer variable can be conditionally defined to one of several constant addresses, initialization must be delayed until run time. The technique is different for near pointers than for far pointers, as shown here:

```
Msg1    BYTE    "String1"
Msg2    BYTE    "String2"
npMsg   NPBYTE  ?
fpMsg   FPBYTE  ?
        .
        .
        .
        mov     npMsg, OFFSET Msg1              ; Load near pointer

        mov     WORD PTR fpMsg[0], OFFSET Msg2  ; Load far offset
        mov     WORD PTR fpMsg[2], SEG Msg2     ; Load far segment
```

If you know that the segment for a far pointer is in a register, you can load it directly:

```
        mov     WORD PTR fpMsg[2], ds           ; Load segment of
                                                ;   far pointer
```

Dynamic Addresses

Often a pointer must point to a dynamic address, meaning the address depends on a run-time condition. Typical situations include memory allocated by MS-DOS (see "Interrupt 21h Function 48h" in Help) and addresses found by the **SCAS** or **CMPS** instructions (see "Processing Strings" in Chapter 5). The following illustrates the technique for saving dynamic addresses:

```
; Dynamically allocated buffer
fpBuf   FPBYTE  0               ; Initialize so offset will be zero
        .
        .
        .
        mov     ah, 48h         ; Allocate memory
        mov     bx, 10h         ; Request 16 paragraphs
        int     21h             ; Call DOS
        jc      error           ; Return segment in AX
        mov     WORD PTR fpBuf[2], ax   ; Load segment
        .                               ;   (offset is already 0)
        .
        .
error:                          ; Handle error
```

Copying Pointers

Sometimes one pointer variable must be initialized by copying from another. Here are two ways to copy a far pointer:

```
fpBuf1  FPBYTE  ?
fpBuf2  FPBYTE  ?
        .
        .
        .
; Copy through registers is faster, but requires a spare register
        mov     ax, WORD PTR fpBuf1[0]
        mov     WORD PTR fpBuf2[0], ax
        mov     ax, WORD PTR fpBuf1[2]
        mov     WORD PTR fpBuf2[2], ax

; Copy through stack is slower, but does not use a register
        push    WORD PTR fpBuf1[0]
        push    WORD PTR fpBuf1[2]
        pop     WORD PTR fpBuf2[2]
        pop     WORD PTR fpBuf2[0]
```

Pointers as Arguments

Most high-level-language procedures and library functions accept arguments passed on the stack. “Passing Arguments on the Stack” in Chapter 7 covers this subject in detail. A pointer is passed in the same way as any other variable, as this fragment shows:

```
; Push a far pointer (segment always pushed first)
        push    WORD PTR fpMsg[2]       ; Push segment
        push    WORD PTR fpMsg[0]       ; Push offset
```

Pushing an address has the same result as pushing a pointer to the address:

```
; Push a far address as a far pointer
        mov     ax, SEG fVar    ; Load and push segment
        push    ax
        mov     ax, OFFSET fVar ; Load and push offset
        push    ax
```

On the 80186 and later processors, you can push a constant in one step:

```
        push    SEG fVar        ; Push segment
        push    OFFSET fVar     ; Push offset
```

Loading Addresses into Registers

Loading a near address into a register (or a far address into a pair of registers) is a common task in assembly-language programming. To reference data pointed to by a pointer, your program must first place the pointer into a register or pair of registers.

Load far addresses as *segment***:***offset* pairs. The following pairs have specific uses:

Segment:Offset Pair	Standard Use
DS:SI	Source for string operations
ES:DI	Destination for string operations
DS:DX	Input for certain DOS functions
ES:BX	Output from certain DOS functions

Addresses from Data Segments

For near addresses, you need only load the offset; the segment is assumed as SS for stack-based data and as DS for other data. You must load both segment and offset for far pointers.

Here is an example of loading an address into DS:BX from a near data segment:

```
        .DATA
Msg     BYTE    "String"
        .
        .
        .
        mov     bx, OFFSET Msg  ; Load address to BX
                                ;   (DS already loaded)
```

Far data can be loaded like this:

```
.FARDATA
Msg     BYTE    "String"
        .
        .
        .
        mov     ax, SEG Msg     ; Load address to ES:BX
        mov     es, ax
        mov     bx, OFFSET Msg
```

You can also read a far address from a pointer in one step, using the **LES** and **LDS** instructions described next.

Far Pointers

The **LES** and **LDS** instructions load a far pointer into a segment pair. The instructions copy the pointer's low word into either ES or DS, and the high word into a given register. The following example shows how to load a far pointer into ES:DI:

```
OutBuf  BYTE    20 DUP (0)

fpOut   FPBYTE  OutBuf
        .
        .
        .
        les     di, fpOut      ; Load far pointer into ES:DI
```

Stack Variables

The technique for loading the address of a stack variable is significantly different from the technique for loading near addresses. You may need to put the correct segment value into ES for string operations. The following example illustrates how to load the address of a local (stack) variable to ES:DI:

```
Task    PROC
        LOCAL   Arg[4]:BYTE

        push    ss      ; Since it's stack-based, segment is SS
        pop     es      ; Copy SS to ES
        lea     di, Arg ; Load offset to DI
```

The local variable in this case actually evaluates to SS:[BP-4]. This is an offset from the stack frame (described in "Passing Arguments on the Stack," Chapter 7). Since you cannot use the **OFFSET** operator to get the offset of an indirect memory operand, you must use the **LEA** (Load Effective Address) instruction.

Direct Memory Operands

To get the address of a direct memory operand, use either the **LEA** instruction or the **MOV** instruction with **OFFSET**. Though both methods have the same effect, the **MOV** instruction produces smaller and faster code, as shown in this example:

```
        lea     si, Msg        ; Four byte instruction
        mov     si, OFFSET Msg ; Three byte equivalent
```

Copying Between Segment Pairs

Copying from one register pair to another is complicated by the fact that you cannot copy one segment register directly to another. Two copying methods are shown here. Timings are for the 8088 processor.

```
; Copy DS:SI to ES:DI, generating smaller code
        push    ds              ; 1 byte, 14 clocks
        pop     es              ; 1 byte, 12 clocks
        mov     di, si          ; 2 bytes, 2 clocks

; Copy DS:SI to ES:DI, generating faster code
        mov     di, ds          ; 2 bytes, 2 clocks
        mov     es, di          ; 2 bytes, 2 clocks
        mov     di, si          ; 2 bytes, 2 clocks
```

Model-Independent Techniques

Often you may want to write code that is memory-model independent. If you are writing libraries that must be available for different memory models, you can use conditional assembly to handle different sizes of pointers. You can use the predefined symbols **@DataSize** and **@Model** to test the current assumptions.

You can use conditional assembly to write code that works with pointer variables that have no specified distance. The predefined symbol **@DataSize** tests the pointer size for the current memory model:

```
Msg1    BYTE    "String1"
pMsg    PBYTE   ?
        .
        .
        .
        IF      @DataSize                       ; @DataSize > 0 for far
        mov     WORD PTR pMsg[0], OFFSET Msg1   ; Load far offset
        mov     WORD PTR pMsg[2], SEG Msg1      ; Load far segment
        ELSE                                    ; @DataSize = 0 for near
        mov     pMsg, OFFSET Msg1               ; Load near pointer
        ENDIF
```

In the following example, a procedure receives as an argument a pointer to a word variable. The code inside the procedure uses **@DataSize** to determine whether the current memory model supports far or near data. It loads and processes the data accordingly:

```
; Procedure that receives an argument by reference
mul8    PROC    arg:PTR WORD

        IF      @DataSize
        les     bx, arg     ; Load far pointer to ES:BX
        mov     ax, es:[bx] ; Load the data pointed to
        ELSE
        mov     bx, arg     ; Load near pointer to BX (assume DS)
        mov     ax, [bx]    ; Load the data pointed to
        ENDIF
        shl     ax, 1       ; Multiply by 8
        shl     ax, 1
        shl     ax, 1
        ret
mul8    ENDP
```

If you have many routines, writing the conditionals for each case can be tedious. The following conditional statements automatically generate the proper instructions and segment overrides.

```
; Equates for conditional handling of pointers
        IF @DataSize
lesIF   TEXTEQU    <les>
ldsIF   TEXTEQU    <lds>
esIF    TEXTEQU    <es:>
        ELSE
lesIF   TEXTEQU    <mov>
ldsIF   TEXTEQU    <mov>
esIF    TEXTEQU    <>
        ENDIF
```

Once you define these conditionals, you can use them to simplify code that must handle several types of pointers. This next example rewrites the above **mul8** procedure to use conditional code.

```
mul8    PROC    arg:PTR WORD

        lesIF   bx, arg         ; Load pointer to BX or ES:BX
        mov     ax, esIF [bx]   ; Load the data from [BX] or ES:[BX]
        shl     ax, 1           ; Multiply by 8
        shl     ax, 1
        shl     ax, 1
        ret
mul8    ENDP
```

The conditional statements from these examples can be defined once in an include file and used whenever you need to handle pointers.