Chapter 9: The Assembly Process

The job of an assembler is to turn assembly language source code into object code. With simpler computer systems this object code is machine language, ready to be loaded into memory and executed. With more complex systems, object code produced by the assembler must be "fixed up" by a linker and/or loader before it can be executed. The first section of this chapter describes the assembly process for a typical assembler and gives some details particular to the Microsoft Macro Assembler. The second section is very specific to the 80x86 microprocessor family; it details the structure of its machine language. The third and fourth sections discuss macros and conditional assembly, respectively. Most assemblers have these capabilities, and these sections describe how MASM implements them. The final section describes the macros in the header file IO.H.

Chapter 9: The Assembly Process

9.1 Two-Pass and One-Pass Assembly

One of the many reasons for writing assembly language rather than machine language is

- ✓ assemblers allow the use of identifiers or symbols to reference data in the data segment and
- ✓ instructions in the code segment. To code in machine language, a programmer must know run-time addresses for data and instructions.
- ✓ An assembler maintains a symbol table that associates each identifier with various attributes.

(One attribute is a location, typically relative to the beginning of a segment, but sometimes an absolute address to be used at run time. Another attribute is the type of the symbol, where possible types include labels for data or instructions, symbols equated to constants, procedure names, macro names, and segment names.)

Some assemblers start assembling a source program with a symbol table that includes all the mnemonics for the language, all register names, and other symbols with reserved usage.

Chapter 9: The Assembly Process

9.1 Two-Pass and One-Pass Assembly

The other main job of an assembler is to output object code that is close to the machine language executed when a program is run.

- ✓ A two-pass assembler scans the source code once to produce a symbol table and a second time to produce the object code.
- ✓ A one-pass assembler only scans the source code one time, but often must patch the object code produced during this scan.

A simple example shows why: If the segment

```
        jmp     endLoop
        add     eax,ecx
endLoop:
```

is scanned, the assembler finds a forward reference to *endLoop* in the jmp instruction. At this point the assembler cannot tell the address of *endLoop*, much less whether this destination is short or near. Clearly the final code must wait at least until the assembler reaches the source code line with the *end-Loop* label.

Chapter 9: The Assembly Process

9.1 Two-Pass and One-Pass Assembly

Typical assemblers use two passes, and some actually use three or more passes. The Microsoft Macro Assembler is a one-pass assembler. This book will not attempt to cover details of how it fixes up object code. You can see part of MASM's symbol table by looking at the end of an assembly listing.

If a symbol is a label for data, then the symbol table may include the size of the data. For instance, the program in Fig. 3.1 contains the directive

```
number2     DWORD ?
```

and the corresponding line in the listing file (Fig. 3.7) is

```
number2 . . . . . . . . . . . . Dword 00000004 _DATA
```

This shows that the size of *number2* has been recorded as a doubleword. Having the size recorded enables MASM to detect incorrect usage of a symbol.

Chapter 9: The Assembly Process

9.1 Two-Pass and One-Pass Assembly

with this definition of *number2*, MASM would indicate an error for the instruction

```
mov     bh,number2
```

since the BH register is byte size while the symbol table identifies *number2* as doubleword size. In addition to the size, if a symbol is associated with multiple objects, a symbol table may contain the number of objects or the total number of bytes associated with the symbol.

If a symbol is equated to a value, then the value is usually stored in the symbol table. When the assembler encounters the symbol in subsequent code, it substitutes the value recorded in the symbol table. In the example program, the source code line

```
cr      EQU 0dh                    ;carriage return character
```

is reflected in the listing file line

```
cr . . . . . . . . . . . . . . . Number 0000000Dh
```

Chapter 9: The Assembly Process

9.1 Two-Pass and One-Pass Assembly

If a symbol is a label for data or an instruction, then its location is entered in the symbol table. An assembler keeps a location counter to compute this value. With a typical assembler, the location counter is set to zero at the beginning of a program or at the beginning of each major subdivision of the program. The Microsoft Macro Assembler sets the location counter to zero at the beginning of each segment. As an assembler scans source code, the location of each datum or instruction is the value of the location counter *before* the statement is assembled. The number of bytes required by the statement is added to the location counter to give the location of the *next* statement. Again looking at the line

```
number2 DWORD ?
```

the listing file shows

```
number2 . . . . . . . . . . . . Dword 00000004 _DATA
```

with 00000004 in the *Value* column. This is the value of the location counter at the time *number2* is encountered in the data segment. The value is 00000004 since the only item preceding *number2* was *number1*, and it took four bytes.

Chapter 9: The Assembly Process

9.1 Two-Pass and One-Pass Assembly

The location counter is used the same way when instructions are assembled. Suppose that the location counter has value 0000012E when MASM reaches the code fragment shown in Fig. 9.1. The location for the symbol *while1* will be 0000012E. The cmp instruction requires three bytes of object code.

Therefore the location counter will have value 00000131 when MASM reaches the jnle instruction. The jnle instruction requires two bytes of object code, so the location counter will increase to 00000133 for the first add instruction. The first add instruction takes two bytes of object code, so the location counter is 00000135 when MASM reaches the second add instruction. Three bytes are required for add ebx,2 so the location counter is 00000138 for the inc instruction. The inc instruction takes a single byte, so the location counter is 00000139 for the jmp instruction. The jmp instruction requires two bytes, making the location counter 0000013B when the assembler reaches the label *endWhile1*. Therefore 0000013B is recorded in the symbol table as the location of *endWhile1*.

Chapter 9: The Assembly Process

9.1 Two-Pass and One-Pass Assembly

```
while1:    cmp    ecx,100       ;count <= 100 ?
           jnle   endWhile1     ;exit if not
           add    eax,[ebx]     ;add value to sum
           add    ebx,4         ;address of next value
           inc    ecx           ;add 1 to count
           jmp    while1
 endWhile1:
```

Figure 9.1 Code with forward references

Chapter 9: The Assembly Process

9.1 Two-Pass and One-Pass Assembly

The location of a symbol is needed for a variety of purposes. Suppose that MASM encounters the statement

```
        mov     eax,number
```

where *number* is the label on a DWORD directive in the data section.

Since the addressing mode for *number* is direct, the assembler needs the offset of *number* for the object code;

this offset is precisely the location of *number* recorded in the symbol table.

Chapter 9: The Assembly Process

9.1 Two-Pass and One-Pass Assembly

The primary job of an assembler is to generate object code. However, a typical assembler does many other tasks. One duty is to reserve storage. A statement like

```
    WORD    20      DUP(?)
```

sets aside 20 words of storage. This storage reservation is typically done one of two ways:

- ✓ **the assembler may write 40 bytes with some known value (like 00) to the object file, or**
- ✓ **the assembler may insert a command that ultimately causes the loader to skip 40 bytes when the program is loaded into memory**

In the latter case, storage at run time will contain whatever values are left over from execution of other programs.

Chapter 9: The Assembly Process

9.1 Two-Pass and One-Pass Assembly

In addition to reserving storage, assemblers can initialize the reserved memory with specified values. The MASM statement

```
WORD   10, 20, 30
```

not only reserves three words of storage, it initializes the first to 000A, the second to 0014 and the third to 001E.

Initial values may be expressed in a variety of ways using MASM assembler. Numbers may be given in different number systems, often binary, octal, decimal, and hexadecimal. The assembler converts character values to corresponding ASCII or EBCDIC character codes.

Assemblers usually allow expressions as initial values. The Microsoft Macro Assembler is typical in accepting expressions that are put together with addition, subtraction, negation, multiplication, division, not, and, or, exclusive or, shift, and relational operators.

Such an expression is evaluated *at assembly time*, producing the value that is actually used in the object code.

Chapter 9: The Assembly Process

9.1 Two-Pass and One-Pass Assembly

Most assemblers can produce a listing file that shows the original source code and some sort of representation of the corresponding object code. Another responsibility of an assembler is to produce error messages when there are errors in the source code.

Rudimentary assemblers just display a line number and an error code for each error.

Most assemblers can include an error message in the listing file at the point where the error occurs. The Microsoft Macro Assembler includes messages in the optional listing file and also displays them on the console.

In addition to the listing that shows source and object code, an assembler often can generate a listing of symbols used in the program. Such a listing may include information about each symbol's attributes—taken from the assembler's symbol table—as well as cross references that indicate the line where the symbol is defined and each line where it is referenced.

Chapter 9: The Assembly Process

9.1 Two-Pass and One-Pass Assembly

One file can reference objects in another. Recall that the EXTRN directive facilitates this for MASM. A linker combines separate object code files into a single file. If one file references objects in the other, the linker changes the references from "to be determined" to locations in the combined file.

Most assemblers produce object code that is relocatable; that is, it can be loaded at any address. One way to do this is to put a map in the object code file that records each place in the program where an address must be modified. Address modifications are usually carried out by the loader. The loader finally produces true machine language, ready for execution.

Another way to get relocatable code is to write it with only relative references; that is, so that each instruction only references an object at some distance from itself, not at a fixed address. In an 80x86 system, most jump instructions are relative, so if a programmer stores data in registers or on the stack, it is fairly easy to produce such a program.

Chapter 9: The Assembly Process

9.1 Two-Pass and One-Pass Assembly

With MASM, a programmer can actually directly reference the location counter using the $ symbol. The code fragment from Fig. 9.1 could be rewritten as

```
cmp     ecx,100        ;count <= 100 ?
jnle    $+10           ;exit if not
add     eax,[ebx]      ;add value to sum
add     ebx,4          ;address of next value
inc     ecx            ;add 1 to count
jmp     $-11
```

This works since the value of the location counter $ is the location of the *beginning* of the jnle statement as it is assembled. Its two bytes and the eight bytes of the next four statements need to be skipped to exit the loop. Similarly the backward reference must skip the inc statement and the four other statements back through the beginning of the cmp statement, a total of eleven bytes.

Chapter 9: The Assembly Process

9.2 80x86 Instruction Coding

This section describes the structure of 80x86 machine language. From this information one could almost assemble an 80x86 assembly language program by hand. However, the primary purpose here is to acquire a better understanding of the capabilities and limitations of the 80x86 microprocessor family.

An 80x86 instruction consists of several fields, which are summarized in Fig. 9.2. Some instructions have only an opcode, while others require that other fields be included. Any included fields always appear in this order.

Chapter 9: The Assembly Process

9.2 80x86 Instruction Coding

Field	Number of bytes	purpose
instruction prefix	0 or 1	$F3_{16}$ for REP, REPE, or REPZ $F2_{16}$ for REPNE or REPNZ $F0_{16}$ for LOCK
address size	0 or 1	value 67_{16} if present; indicates that a displacement is a 16-bit address rather than the default 32-bit size
operand size	0 or 1	value 66_{16} if present; indicates that a memory operand is 16-bit if in 32-bit mode or 32 bit if in 16-bit mode
segment override	0 or 1	indicates that an operand is in a segment other than the default segment
opcode	1 or 2	operation code
mod-reg-r/m	0 or 1	indicates register or memory operand, encodes register(s)
scaled index base byte	0 or 1	additional scaling and register information
Displacement	0 to 4	an address
immediate	0 to 4	an immediate value

Figure 9.2 80x86 instruction field

Chapter 9: The Assembly Process

9.2 80x86 Instruction Coding

The repeat prefixes for string instructions were discussed in Chapter 7. There you learned that adding a repeat prefix to one of the basic string instructions effectively changes it into a new instruction that automatically iterates a basic operation. The repeat prefix is coded in the instruction prefix byte, with the opcode of the basic string instruction in the opcode byte. Repeat prefix bytes can be coded only with the basic string instructions.

The LOCK prefix is not illustrated in this book's code. It can be used with a few selected instructions and causes the system bus to be locked during execution of the instruction. Locking the bus guarantees that the 80x86 processor has exclusive use of shared memory.

All the code in this book uses 32-bit memory addresses. In a 32-bit address environment it is possible to have an instruction that only contains a 16-bit address. When an address size byte of 67_{16} is coded, a two-byte rather than a four-byte displacement is used in the displacement field.

Chapter 9: The Assembly Process

9.2 80x86 Instruction Coding

All the code in this book uses 32-bit memory addresses. In a 32-bit address environment it is possible to have an instruction that only contains a 16-bit address. When an address size byte of 67_{16} is coded, a two-byte rather than a four-byte displacement is used in the displacement field.

The 80x86 CPU has a status bit that determines whether operands are 16-bit or 32-bit. With the assembly and linking options we have used, that bit is always set to indicate 32-bit operands. Each time you code a word-size operand, the generated instruction includes the 66_{16} prefix byte to indicate the 16-bit operand.

What indicates a byte-size operand? A different opcode. Why don't 16-bit and 32-bit operands use distinct opcodes? This design decision was made by Intel. The original 8086 processor design had 16-bit registers and used separate opcodes for 8-bit and 16-bit operand sizes; no instruction used 32-bit operands. When the 80386 was designed with 32-bit registers, the choice was made to "share" opcodes for 16-bit and 32-bit operand sizes rather than to introduce many new opcodes.

Chapter 9: The Assembly Process

9.2 80x86 Instruction Coding

The *mod-reg-r/m* byte has different uses for different instructions. When present it always has three fields, a two-bit *mod* field (for "mode"), a three-bit *reg* field (for "register," but sometimes used for other purposes), and a 3-bit *r/m* field (for "register/memory"). The *mod-reg-r/m* byte is examined below.

The opcode field completely identifies many instructions, but some require additional information-for example, to determine the type of operand or even to determine the operation itself. You have previously seen the latter situation.

For example, each of the instructions add, or, adc, sbb, and, sub, xor, and cmp having a byte-size operand in a register or memory and an immediate operand uses the opcode 80. Which of these eight instructions is determined by the *reg* field of the *mod-reg-r/m* byte. For the particular case of the 80 opcode, the *reg* field is 000 for add, 001 for or, 010 for adc, 011 for sbb, 100 for and, 101 for sub, 110 for xor, and 111 for cmp.

Chapter 9: The Assembly Process

9.2 80x86 Instruction Coding

The opcode 80 is one of twelve in which the *reg* field of the *mod-reg-r/m* byte actually determines the instruction. The others are 81, 82, 83, D0, D1, D2, D3, F6, F7, FE, and FF. The table in Fig. 9.3 gives *reg* field information for the most common instructions.

		reg field							
		000	001	010	011	100	101	110	111
Opcode	80, 81, 82, 83	ADD	OR	ADC	SBB	AND	SUB	XOR	CMP
	D0, D1, D2, D3	ROL	ROR	RCL	RCR	SHL	SHR		SAR
	F6, F7	TEST		NOT	NEG	MUL	IMUL	DIV	IDIV
	FE, FF	INC	DEC					PUSH	

Figure 9.3 reg field for specified opcodes

Chapter 9: The Assembly Process

9.2 80x86 Instruction Coding

Each two-operand, nonimmediate 80x86 instruction has at least one register operand. The *reg* field contains a code for this register. Figure 9.4 shows how the eight possible register codes are assigned.

The meaning of a *reg* code varies with the operand size and with the instruction, so that, for example, the same code is used for ECX and CL. These codes are used any time information about a register is encoded in an instruction, whether in the *reg* field or other places.

Reg code	**Register 32**	**Register 16**	**Register 8**	**Segment Register**
000	EAX	AX	AL	ES
001	ECX	CX	CL	CS
010	EDX	DX	DL	SS
011	EBX	BX	BL	DS
100	ESP	SP	AH	FS
101	EBP	BP	CH	GS
110	ESI	SI	DH	
111	EDI	DI	BH	

Figure 9.4 80x86 register codes

Chapter 9: The Assembly Process

9.2 80x86 Instruction Coding

The *mod* field is also used to determine the type of operands an instruction has. Often the same opcode is used for an instruction that has two register operands or one register operand and one memory operand. The choice *mod*=11 means that the instruction is a register-to-register operation or an immediate-to-register operation. For a register-to-register operation, the destination register is coded in the *reg* field and the source register is coded in the *r/m* field. Both use the register codes shown in Fig. 9.4. For an immediate-to-register operation, the operation is coded as shown in Fig. 9.3 and the destination register is coded in the *r/m* field. The situation is complicated for the other possible *mod* values and depends on the *r/m* field as well as the *mod* field. For *r/m*=100, it also depends on the scaled index base (*SIB*) byte.

The *SIB* byte consists of three fields, a two-bit scaling field, a three-bit index register field, and a three-bit base register field.

The scale values are 00 for 1, 01 for 2, 10 for 4, and 11 for 8.

Chapter 9: The Assembly Process

9.2 80x86 Instruction Coding

The index and base register encodings are as shown in Fig. 9.4, except that 100 cannot appear in the index register field since ESP cannot be an index register. Figure. 9.5 shows the different encodings. The *mod* field in these formats tells how many bytes there are in the displacement. A value of 00 means that there is no displacement in the machine code, except when *r/m=101* when there is *only* a displacement. This special case is for direct memory addressing, so is frequently used. A *mod* value of 01 means that there is a displacement *byte* in the machine code; this byte is treated as a signed number and is extended to a doubleword before it is added to the value from the base register and/or index register. A value of 10 means that there is a displacement *doubleword* in the machine code; this doubleword is added to the value that comes from the base register and/or scaled index register. The scaling factor is multiplied times the value in the index register.

Chapter 9: The Assembly Process

9.2 80x86 Instruction Coding

Many

Chapter 9: The Assembly Process

9.3 Macro Definition and Expansion

Macros

A macro is shorthand for a sequence of other statements. The assembler expands a macro to the statements it represents, and then assembles these new statements.

A macro definition resembles a procedure definition in a high-level language. The first line gives the name of the macro being defined and a list of parameters; the main part of the definition consists of a collection of statements.

A macro is called much like a high-level language procedure, too; the name of the macro is followed by a list of arguments.

These similarities are superficial. A procedure call in a high-level language is generally compiled into a sequence of instructions to push parameters on the stack followed by a call instruction, whereas a macro call actually expands into statements given in the macro, with the arguments substituted for the parameters used in the macro definition. Code in a macro is repeated every time a macro is called, but there is just one copy of the code for a procedure.

Chapter 9: The Assembly Process

9.3 Macro Definition and Expansion

Every macro definition is bracketed by MACRO and ENDM directives. The format of a macro definition is

```
name    MACRO list of parameters
        assembly language statements
        ENDM
```

The parameters in the MACRO directive are ordinary symbols, separated by commas. The assembly language statements may use the parameters as well as registers, immediate operands, or symbols defined outside the macro.

A macro definition can appear anywhere in an assembly language source code file.

Chapter 9: The Assembly Process

9.3 Macro Definition and Expansion

Suppose that a program design requires several pauses where the user is prompted to press the [Enter] key. Rather than write this code every time or use a procedure, a macro *pause* can be defined. Figure 9.7 gives such a definition.

```
pause       MACRO
;prompt user and wait for [Enter] to be pressed
            output pressMsg        ;"Press [Enter]"
            input stringIn,5       ;input
            ENDM
```

Figure 9.7 pause Macro

The *pause* macro has no parameter, so a call expands to almost exactly the same statements as are in the definition.

Chapter 9: The Assembly Process

9.3 Macro Definition and Expansion

If the statement

```
pause
```

is included in subsequent source code, then the assembler expands this macro call into the statements

```
output      pressMsg        ;"Press [Enter]"
input       stringIn,5      ;input
```

Of course, each of these statements is itself a macro call and will expand to additional statements. Notice that the *pause* macro is not self-contained; it references two fields in the data segment:

```
pressMsg    BYTE    "Press [Enter] to continue", 0
stringIn    BYTE    5 DUP (?)
```

Note again that the definition and expansion for the pause macro contain no **ret** statement.

Chapter 9: The Assembly Process

9.3 Macro Definition and Expansion

Figure 9.8 gives a definition of a macro *add2* that finds the sum of two parameters, putting the result in the EAX register. The parameters used to define the macro are *nbr1* and *nbr2*. These labels are local to the definition. The same names could be used for other purposes in the program, although some human confusion might result.

```
add2    MACRO   nbr1,nbr2
;put sum of two doubleword parameters in EAX
mov     eax,nbr1
add     eax,nbr2
        ENDM
```

Figure 9.7 pause Macro

Chapter 9: The Assembly Process

9.3 Macro Definition and Expansion

The statements to which *add2* expands depends on the arguments used in a call. For example, the macro call

```
        add2    value,30                ;value + 30
```

expands to

```
        ;put sum of two doubleword parameters in EAX
        mov     eax,value
        add     eax, 30
```

The statement

```
        add2    value1,value2           ;value1 + value2
```

expands to

```
        ;put sum of two doubleword parameters in EAX
        mov     eax,value1
        add     eax,value2
```

Chapter 9: The Assembly Process

9.3 Macro Definition and Expansion

The macro call

```
add2    eax,ebx         ;sum of two values
```

expands to

```
;put sum of two doubleword parameters in EAX
mov     eax,eax
add     eax,ebx
```

The instruction **mov eax,eax** is legal, even if it accomplishes nothing.

In each of these examples, the first argument is substituted for the first parameter *nbr1* and the second argument is substituted for the second parameter *nbr2*.

Each macro results in two mov instructions, but since the types of arguments differ, the object code will vary.

Chapter 9: The Assembly Process

9.3 Macro Definition and Expansion

If one of the parameters is missing the macro will still be expanded. For instance, the statement

```
add2    value
```

expands to

```
;put sum of two doubleword parameters in EAX
mov     eax,value
add     eax,
```

The argument *value* replaces *nbr1* and an empty string replaces *nbr2*. The assembler will report an error, but it will be for the illegal add instruction that results from the macro expansion, not directly because of the missing argument.

Chapter 9: The Assembly Process

9.3 Macro Definition and Expansion

Similarly, the macro call

```
        add     ,value
```

expands to

```
        ;put sum of two doubleword parameters in EAX
        mov     eax,
        add     eax,value
```

The comma in the macro call separates the first missing argument from the second argument *value*. An empty argument replaces the parameter *nbr1*. The assembler will again report an error, this time for the illegal mov instruction.

Chapter 9: The Assembly Process
9.3 Macro Definition and Expansion

Figure 9.9 shows the definition of a macro *swap* that will exchange the contents of two doublewords in memory. It is very similar to the 80x86 **xchg** instruction that will not work with two memory operands.

```
swap    MACRO   dword1,dword2
;exchange two doublewords in memory
push    eax
mov     eax,dword1
xchg    eax,dword2
mov     dword1,eax
pop     eax
ENDM
```

Figure 9.9 Macro to swap two memory words

Chapter 9: The Assembly Process

9.3 Macro Definition and Expansion

As with the *add2* macro, the code generated by calling the *swap* macro depends on the arguments used. For example, the call

```
swap    [ebx],[ebx+4]  ;swap adjacent words in array
```

expands to

```
;exchange two doublewords in memory
push    eax
mov     eax,[ebx]
xchg    eax,[ebx+4]
mov     [ebx],eax
pop     eax
```

It might not be obvious to the user that the swap macro uses the EAX register, so the push and pop instructions in the macro protect the user from accidentally losing the contents of this register.

Chapter 9: The Assembly Process

9.3 Macro Definition and Expansion

Figure 9.10 gives a definition of a macro *min2*, which finds the minimum of two doubleword signed integers, putting the smaller in the EAX register.

```
min2    MACRO   first,second
        LOCAL endIfMin
;put smaller of two doublewords in the EAX register
        mov     eax,first
        cmp     eax,second
        jle     endIfMin
        mov     eax,second
endIfMin:
        ENDM
```

Figure 9.10 Macro to find smaller of two memeory words

Chapter 9: The Assembly Process

9.3 Macro Definition and Expansion

The LOCAL directive is used only within a macro definition and must be the first statement after the MACRO directive. (Not even a comment can separate the MACRO and LOCAL directives.) It lists one or more symbols, separated by commas, which are used within the macro definition.

Each time the macro is expanded and one of these symbols is needed, it is replaced by a symbol starting with two question marks and ending with four hexadecimal digits (??0000, ??0001, etc.) The same *??dddd* symbol replaces the local symbol each place the local symbol is used in one particular expansion of a macro call.

Chapter 9: The Assembly Process

9.3 Macro Definition and Expansion

The macro call

```
        min2    [ebx],ecx       ;find smaller of two values
```

might expand to the code

```
        LOCAL   endIfMin
        ;put smaller of two doublewords in the EAX register
        mov     eax,[ebx]
        cmp     eax,ecx
        jle     ??000C
        mov     eax,ecx
??000C:
```

Here *endIfMin* has been replaced the two places it appears within the macro definition by ??000C in the expansion. Another expansion of the same macro would use a different number after the question marks.

Chapter 9: The Assembly Process

9.3 Macro Definition and Expansion

The MASM assembler has several directives that control how macros and other statements are shown in .LST files. The most useful are

- ✓ **`.LIST` that causes statements to be included in the listing file**
- ✓ **`.NOLIST` that completely suppresses the listing of all statements, and**
- ✓ **`.NOLISTMACRO` that selectively suppresses macro expansions while allowing the programmer's original statements to be listed**

The file IO.H ends starts with a .NOLIST directive so that macro definitions do not clutter the listing. Similarly IO.H ends with .NOLISTMACRO and .LIST directives so that macro expansion listings do not obscure the programmer's code, but original statements are listed.

Chapter 9: The Assembly Process

9.4 Conditional Assembly

The Microsoft Macro Assembler can observe various conditions that can be tested at assembly time and alter how the source code is assembled on the basis of these conditions.

For instance, a block of code may be assembled or skipped based on the definition of a constant. This ability to do conditional assembly is especially useful in macro definitions.

For example, two macros using the same mnemonic may be expanded into different sequences of statements based on the number of operands present.

Chapter 9: The Assembly Process

9.4 Conditional Assembly

Figure 9.11 shows a definition for a macro *addAll* that will add one to five doubleword integers, putting the sum in the EAX register. It employs the conditional assembly directive IFNB ("if not blank").

```
addAll    MACRO nbr1, nbr2, nbr3, nbr4, nbr5
;add up to 5 doubleword integers, putting sum in EAX
mov       eax,nbr1            ;first operand
IFNB      <nbr2>
add       eax,nbr2            ;second operand
ENDIF
IFNB      <nbr3>
add       eax,nbr3            ;third operand
ENDIF
IFNB      <nbr4>
add       eax,nbr4            ;fourth operand
ENDIF
IFNB      <nbr5>
add       eax,nbr5            ;fifth operand
ENDIF
ENDM
```

Figure 9.11 addall macro using conditional assembly

Chapter 9: The Assembly Process

9.4 Conditional Assembly

Given the macro call

```
        addAll  ebx,ecx,edx,number,1
```

each of the five macro parameters has a corresponding argument, so the macro expands to

```
        mov     eax,ebx         ;first operand
        add     eax,ecx         ;second operand
        add     eax,edx         ;third operand
        add     eax,number      ;fourth operand
        add     eax,1           ;fifth operand
```

Chapter 9: The Assembly Process

9.4 Conditional Assembly

The macro call

```
addAll  ebx,ecx,45     ;value1 + value2 + 45
```

has only three arguments. The argument ebx becomes the value for parameter *nbr1*, ecx is substituted for *nbr2*, and 45 will be used for *nbr3*, but the parameters *nbr4* and *nbr5* will be blank. Therefore the macro expands to the statements

```
mov     eax,ebx        ;first operand
add     eax,ecx        ;second operand
add     eax,45         ;third operand
```

Although it would be unusual to do so, arguments other than trailing ones can be omitted.

Chapter 9: The Assembly Process

9.4 Conditional Assembly

For example, the macro call

```
        addAll  ebx, ,ecx
```

has ebx corresponding to *nbr1* and ecx matched to *nbr3*, but all other parameters will be blank. Therefore the macro expands to

```
        mov     eax,ebx         ;first operand
        add     eax,ecx         ;third operand
```

If the first argument is omitted in an *addAll* macro call, the macro will still be expanded. However, the resulting statement sequence will contain a mov instruction with a missing operand, and this statement will cause MASM to issue an error message.

Chapter 9: The Assembly Process

9.4 Conditional Assembly

For example, the macro call

```
        addAll  ,value1,value2
```

expands to

```
        mov     eax,            ;first operand
        add     eax,value1      ;second operand
        add     eax,value2      ;third operand
```

Chapter 9: The Assembly Process

9.4 Conditional Assembly

An unusual use of the *addAll* macro is illustrated by the call

```
addAll  value, eax, eax, value, eax ; 10 * value
```

that expands to

```
mov     eax,value               ;first operand
add     eax,eax                 ;second operand
add     eax,eax                 ;third operand
add     eax,value               ;fourth operand
add     eax,eax                 ;fifth operand
```

The comment "10 * value" explains the purpose of this call.

Chapter 9: The Assembly Process

9.4 Conditional Assembly

The Microsoft assembler provides several conditional assembly directives. The IFNB directive has a companion IFB ("if blank") that checks if a macro parameter *is* blank.

The IF and IFE directives examine an expression whose value can be determined at assembly time. For IF, MASM assembles conditional code if the value of the expression is not zero. For IFE, MASM includes conditional code if the value is zero.

The IFDEF and IFNDEF are similar to IF and IFE. They examine a symbol and MASM assembles conditional code depending on whether or not the symbol has previously been defined in the program.

Each conditional assembly block is terminated by the ENDIF directive. LSEIF and ELSE directives are available to provide alternative code .

Chapter 9: The Assembly Process
9.4 Conditional Assembly

In general, blocks of conditional assembly code look like

```
IF... [operands]
statements
ELSEIF ...
statements
ELSE
statements
ENDIF
```

Operands vary with the type of IF and are not used with all types. The ELSEIF directive and statements following it are optional, as are the ELSE directive and statements following it. There can be more than one ELSEIF directive, but at most one ELSE directive.

Chapter 9: The Assembly Process
9.4 Conditional Assembly

The above syntax strongly resembles what appears in many high-level languages. It is important to realize, however, that these directives are used at *assembly* time, not at *execution* time.

The EXITM directive can be used to make some macro definitions simpler to write and understand. When MASM is processing a macro call and finds an EXITM directive, it immediately stops expanding the macro, ignoring any statements following EXITM in the macro definition.

The design

```
if  condition
then
     process assembly language statements for condition;
else
     process statements for negation of condition;
end if;
```

Chapter 9: The Assembly Process

9.4 Conditional Assembly

and the alternative design

```
________________________________________________________
if condition
then
     process assembly language statements for condition;
     terminate expansion of macro;
end if;
process statements for negation of condition;
________________________________________________________
```

are equivalent, assuming that no macro definition statements follow those sketched in the designs. These alternative designs can be implemented using

Chapter 9: The Assembly Process

9.4 Conditional Assembly

These alternative designs can be implemented using

```
IF... [operands]
assembly language statements for condition
ELSE
assembly language statements for negation of condition
ENDIF
```

and

```
IF... [operands]
assembly language statements for condition
EXITM
ENDIF
assembly language statements for negation of condition
```

Chapter 9: The Assembly Process

9.4 Conditional Assembly

A macro definition using EXITM appears in Fig. 9.12.

```
          min2      MACRO     value1,value2,extra
          LOCAL     endIfLess
          ;put smaller of value1 and value2 in EAX

          IFB       <value1>
          .ERR <first argument missing in min2 macro>
          EXITM
          ENDIF

          IFB       <value2>
          .ERR <second argument missing in min2 macro>
          EXITM
          ENDIF

          IFNB      <extra>
          .ERR <more than two arguments in min2 macro>
          EXITM
          ENDIF

          mov       ax,value1           ;;first value to EAX
          cmp       eax, value2         ;;value1 <= value2?
          jle       endIfLess           ;;done if so
          mov       eax, value2 ;;otherwise value2 smaller
endIfLess:
          ENDM
```

Figure 9.12: Improved *min2* macro

Chapter 9: The Assembly Process

9.4 Conditional Assembly

.ERR

If assembly errors are eliminated by avoiding generation of illegal statements, a user may not know when a macro call is faulty.

It requires additional effort to inform the user of an error. One way to do this is with the .ERR directive. This directive generates a forced error at assembly time, resulting in a message to the console and a message to the listing file, if any.

It also ensures that no .obj file is produced for the assembly. The .ERR directive is often followed by a string enclosed by < and >. This string is included in the error message.

Chapter 9: The Assembly Process

9.5 Macros in IO.H

Macros in the file IO.H are designed to provide simple, safe access to standard input and output devices. Figure 9.13 shows the contents of IO.H and the remainder of the section discusses the directives and macros in the file.

Most of the file IO.H consists of macro definitions that, when used, generate code to call external procedures. However, the file does contain other directives. It begins with a .NOLIST directive; this suppresses the listing of all source code, in particular the contents of IO.H. It then has EXTRN directives that identify the external procedures called by the macros. The file ends with a .NOLISTMACRO directive to suppress listing of any macro expansions and an .LIST directive so that the user's statements following the directive INCLUDE io.h will again be shown in the listing file.

Chapter 9: The Assembly Process

9.5 Macros in IO.H

The bulk of the file IO.H consists of definitions for *itoa*, *atoi*, *dtoa*, *atod*, *output*, and *input* macros. These definitions have similar structures. Each uses IFB and IFNB directives to check that a macro call has the correct number of arguments. If not, .ERR directives are used to generate forced errors and appropriate messages. Actually, the checks are not quite complete.

Assuming that its arguments are correct, an input/output macro call expands to a sequence of instructions that call the appropriate external procedure, for instance *itoaproc* for the macro *itoa*. Parameters are passed on the stack, but some code sequences use a register to temporarily contain a value, with push and pop instructions to ensure that these registers are not changed following a macro call.

Chapter 9: The Assembly Process

Chapter Summary

This chapter has discussed the assembly process. A typical two-pass assembler scans an assembly language program twice, using a location counter to construct a symbol table during the first pass, and completing assembly during the second pass. The symbol table contains information about each identifier used in the program, including its type, size, and location. Assembly can be done in a single pass if the object code is "fixed up" when forward references are resolved.

A machine instruction may have one or more prefix bytes. However, the main byte of machine code for each 80x86 instruction is its opcode. Some instructions are a single byte long, but most consist of multiple bytes. The next byte often has the format *mod reg r/m* where *reg* indicates a source or destination register, and the other two fields combine to describe the addressing mode. Other instruction bytes contain additional addressing information, immediate data, or the address of a memory operand.

Chapter 9: The Assembly Process

Chapter Summary

Macros are defined using MACRO and ENDM directives. Macros may use parameters that are associated with corresponding arguments in macro calls. A call is expanded at assembly time. The statements in the expansion of a macro call appear in the macro definition, with arguments substituted for parameters. A macro definition may declare local labels that MASM expands to different symbols for different macro calls.

Conditional assembly may be used in regular code or in macro definitions to generate different statements, based on conditions that can be checked at assembly time. The IFB and IFNB directives are used in macros to check for the absence or presence of arguments. Several other conditional assembly directives are also available, including IF, IFE, IFDEF, and IFNDEF. An ELSE directive may be used to provide two alternative blocks of code, and the ENDIF directive ends a conditional assembly block.

Chapter 9: The Assembly Process

Chapter Summary

If the assembler encounters an EXITM directive when expanding a macro definition, it immediately terminates expansion of the macro. The .ERR directive triggers a forced error so that MASM displays an error message and produces no .OBJ file for the assembly.

The file IO.H contains definitions for a collection of input/output macros, and a few directives. These macro definitions use conditional assembly to check for missing or extra arguments and generate code that calls external procedures.