4. ASSEMBLER DIRECTIVES

This chapter describes the assembler directives used to control the 8080/85 assembler in its generation of object code. This chapter excludes the macro directives, which are discussed as a separate topic in Chapter 5.

Generally, directives have the same format as instructions and can be interspersed throughout your program. Assembler directives discussed in this chapter are grouped as follows:

GENERAL DIRECTIVES:

- Symbol Definition

  EQU
  SET

- Data Definition

  DB
  DW

- Memory Reservation

  DS

- Conditional Assembly

  IF
  ELSE
  ENDIF

- Assembler Termination

  END

LOCATION COUNTER CONTROL AND RELOCATION:

- Location Counter Control

  ASEG
  DSEG
  CSEG
  ORG

- Program Linkage

  PUBLIC
  EXTRN
  NAME
  STKLN

Three assembler directives -- EQU, SET, and MACRO -- have a slightly different format from assembly language instructions. The EQU, SET, and MACRO directives require a *name* for the symbol or macro being defined to be present in the label field. Names differ from labels in that they must *not* be terminated with a colon (:) as labels are. Also, the LOCAL and ENDM directives prohibit the use of the label field.

The MACRO, ENDM, and LOCAL directives are explained in Chapter 5.

SYMBOL DEFINITION

The assembler automatically assigns values to symbols that appear as instruction labels. This value is the current setting of the location counter when the instruction is assembled. (The location counters are explained under 'Address Control and Relocation,' later in this chapter.)

You may define other symbols and assign them values by using the EQU and SET directives. Symbols defined using EQU cannot be redefined during assembly; those defined by SET can be assigned new values by subsequent SET directives.

The name required in the label field of an EQU or SET directive must *not* be terminated with a colon.

Symbols defined by EQU and SET have meaning throughout the remainder of the program. This may cause the symbol to have illegal multiple definitions when the EQU or SET directive appears in a macro definition. Use the LOCAL directive (described in Chapter 5) to avoid this problem.

EQU Directive

EQU assigns the value of 'expression' to the name specified in the label field.

Label	*Opcode*	*Operand*
name	EQU	expression

The required name in the label field may not be terminated with a colon. This name cannot be redefined by a subsequent EQU or SET directive. The EQU expression cannot contain any external symbol. (External symbols are explained under 'Location Counter Control and Relocation,' later in this chapter.)

Assembly-time evaluation of EQU expressions always generates a modulo 64K address. Thus, the expression always yields a value in the range 0 - 65535.

Example:

The following EQU directive enters the name ONES into the symbol table and assigns the binary value 11111111 to it:

```
ONES    EQU     OFFH
```

The value assigned by the EQU directive can be recalled in subsequent source lines by referring to its assigned name as in the following IF directive (where TYPE has been previously defined):

```
IF TYPE EQ ONES
.
.
.
ENDIF
```

SET Directive

SET assigns the value of 'expression' to the name specified in the label field.

Label	*Opcode*	*Operand*
name	SET	expression

The assembler enters the value of 'expression' into the symbol table. Whenever 'name' is encountered subsequently in the assembly, the assembler substitutes its value from the symbol table. This value remains unchanged until altered by a subsequent SET directive.

The function of the SET directive is identical to EQU except that 'name' can appear in multiple SET directives in the same program. Therefore, you can alter the value assigned to 'name' throughout the assembly.

Assembly-time evaluation of SET expressions always generates a modulo 64K address. Thus, the expression always yields a value in the range 0 - 65535.

Examples:

Label	*Opcode*	*Operand*	*Assembled Code*
IMMED	SET	5	
	ADI	IMMED	C605
IMMED	SET	10H-6	
	ADI	IMMED	C60A

DATA DEFINITION

The DB (define byte) and DW (define word) directives enable you to define data to be stored in your program. Data can be specified in the form of 8-bit or 16-bit values, or as a string of text characters.

DB Directive

The DB directive stores the specified data in consecutive memory locations starting with the current setting of the location counter.

Label	*Opcode*	*Operands*
optional:	DB	expression(s) or string(s)

Each symbol in the expression(s) must be previously defined. The operand field of the DB directive can contain a list of expressions and/or text strings. The list can contain up to eight total items; list items must be separated by commas. Because of limited workspace, the assembler may not be able to handle a total of eight items when the list includes a number of complex expressions. If you ever have this problem, it is easily solved: simply use two or more directives to shorten the list.

Expressions must evaluate to 1-byte (8-bit) numbers in the range –256 through 255. Text strings can consist of a maximum of 128 ASCII characters enclosed in quotes.

The assembler's relocation feature treats all external and relocatable symbols as 16-bit addresses. When one of these symbols appears in an operand expression of the DB directive, it must be preceded by either the HIGH or LOW operator to specify which byte of the address is to be used in the evaluation of the expression. When neither operator is present, the assembler assumes the LOW operator and issues an error message.

If the optional label is present, it is assigned the starting value of the location counter, and thus references the first byte stored by the DB directive. Therefore, the label STR in the following examples refers to the letter T of the string TIME.

Examples:

Label	*Opcode*	*Operands*	*Assembled Code*
STR:	DB	'TIME'	54494D45
HERE:	DB	0A3H	A3
WORD1:	DB	–03H,5*2	FD0A

DW Directive

The DW directive stores each 16-bit value from the expression list as an address. The values are stored starting at the current setting of the location counter.

Label	*Opcode*	*Operands*
optional:	DW	expression list

Each symbol in the expression list must be previously defined. The least significant eight bits of the first value in the expression list are stored at the current setting of the location counter; the most significant eight bits are stored at the next higher location. This process is repeated for each item in the expression list.

Expressions evaluate to 1–word (16-bit) numbers, typically addresses. If an expression evaluates to a single byte, it is assumed to be the low order byte of a 16-bit word where the high order byte is all zeros.

5. MACROS

INTRODUCTION TO MACROS

Why Use Macros?

A macro is essentially a facility for replacing one set of parameters with another. In developing your program, you will frequently find that many instruction sequences are repeated several times with only certain parameters changed.

As an example, suppose that you code a routine that moves five bytes of data from one memory location to another. A little later, you find yourself coding another routine to move four bytes from a different source field to a different destination field. If the two routines use the same coding techniques, you will find that they are identical except for three parameters: the character count, the source field starting address, and the destination field starting address. Certainly it would be handy if there were some way to regenerate that original routine substituting the new parameters rather than rewrite that code yourself. The macro facility provides this capability and offers several other advantages over writing code repetitiously:

- The tedium of frequent rewrite (and the probability of error) is reduced.
- Symbols used in macros can be restricted so that they have meaning only within the macro itself. Therefore, as you code your program, you need not worry that you will accidentally duplicate a symbol used in a macro. Also, a macro can be used any number of times in the same program without duplicating any of its own symbols.
- An error detected in a macro need be corrected only once regardless of how many times the macro appears in the program. This reduces debugging time.
- Duplication of effort between programmers can be reduced. Useful functions can be collected in a library to allow macros to be copied into different programs.

In addition, macros can be used to improve program readability and to create structured programs. Using macros to segment code blocks provides clear program notation and simplifies tracing the flow of the program.

What Is A Macro?

A macro can be described as a routine *defined* in a formal sequence of prototype instructions that, when *called* within a program, results in the replacement of each such call with a code *expansion* consisting of the actual instructions represented.

The concepts of macro definition, call, and expansion can be illustrated by a typical business form letter, where the prototype instructions consist of preset text. For example, we could define a macro CNFIRM with the text

```
Air Flight welcomes you as a passenger.
Your flight number FNO leaves at DTIME and arrives in DEST at ATIME.
```

This macro has four dummy parameters to be replaced, when the macro is called, by the actual flight number, departure time, destination, and arrival time. Thus the macro call might look like

```
CNFIRM      123, '10:45', 'Ontario', '11:52'
```

A second macro, CAR, could be called if the passenger has requested that a rental car be reserved at the destination airport. This macro might have the text

```
Your automobile reservation has been confirmed with MAKE rent-a-car agency.
```

Finally, a macro GREET could be defined to specify the passenger name.

```
Dear NAME:
```

The entire text of the business letter (source file) would then look like

```
GREET 'Ms. Scannel'
CNFIRM 123, '10:45', 'Ontario', '11:52'
CAR 'Blotz'
We trust you will enjoy your flight.

Sincerely,
```

When this source file is passed through a macro processor, the macro calls are expanded to produce the following letter.

```
Dear Ms. Scannel:

Air Flight welcomes you as a passenger. Your flight number 123 leaves at 10:45 and arrives
in Ontario at 11:52. Your automobile reservation has been confirmed with Blotz rent-a-car
agency.

We trust you will enjoy your flight.

Sincerely,
```

While this example illustrates the substitution of parameters in a macro, it overlooks the relationship of the macro processor and the assembler. The purpose of the macro processor is to generate source code which is then assembled.

Macros Vs. Subroutines

At this point, you may be wondering how macros differ from subroutines invoked by the CALL instruction. Both aid program structuring and reduce the coding of frequently executed routines.

One distinction between the two is that subroutines necessarily branch to another part of your program while macros generate in-line code. Thus, a program contains only one version of a given subroutine, but contains as many versions of a given macro as there are calls for that macro.

Notice the emphasis on 'versions' in the previous sentence, for this is a major difference between macros and subroutines. A macro does not necessarily generate the same source code each time it is called. By changing the parameters in a macro call, you can change the source code the macro generates. In addition, macro parameters can be tested at assembly-time by the conditional assembly directives. These two tools enable a general-purpose macro definition to generate customized source code for a particular programming situation. Notice that macro expansion and any code customization occur at assembly-time and at the source code level. By contrast, a generalized subroutine resides in your program and requires execution time.

It is usually possible to obtain similar results using either a macro or a subroutine. Determining which of these facilities to use is not always an obvious decision. In some cases, using a single subroutine rather than multiple in-line macros can reduce the overall program size. In situations involving a large number of parameters, the use of macros may be more efficient. Also, notice that macros can call subroutines, and subroutines can contain macros.

USING MACROS

The assembler recognizes the following macro operations:

- MACRO directive
- ENDM directive
- LOCAL directive
- REPT directive
- IRP directive
- IRPC directive
- EXITM directive
- Macro call

All of the directives listed above are related to macro definition. The macro call initiates the parameter substitution (macro expansion) process.

Macro Definition

Macros must be defined in your program before they can be used. A macro definition is initiated by the MACRO assembler directive, which lists the *name* by which the macro can later be called, and the *dummy parameters* to be replaced during macro expansion. The macro definition is terminated by the ENDM directive. The prototype instructions bounded by the MACRO and ENDM directives are called the *macro body*.

When label symbols used in a macro body have 'global' scope, multiply-defined symbol errors result if the macro is called more than once. A label can be given limited scope using the LOCAL directive. This directive assigns a unique value to the symbol each time the macro is called and expanded. Dummy parameters also have limited scope.

Occasionally you may wish to duplicate a block of code several times, either within a macro or in line with other source code. This can be accomplished with minimal coding effort using the REPT (repeat block), IRP (indefinite repeat), and IRPC (indefinite repeat character) directives. Like the MACRO directive, these directives are terminated by ENDM.

The EXITM directive provides an alternate exit from a macro. When encountered, it terminates the current macro just as if ENDM had been encountered.

Macro Definition Directives

MACRO Directive

Label	*Opcode*	*Operand*
name	MACRO	optional dummy parameter(s)

The name in the label field specifies the name of the macro body being defined. Any valid user-defined symbol name can be used as a macro name. Note that this name must be present and must *not* be terminated by a colon.

A dummy parameter can be any valid user-defined symbol name or can be null. When multiple parameters are listed, they must be separated by commas. The scope of a dummy parameter is limited to its specific macro definition. If a reserved symbol is used as a dummy parameter, its reserved value is not recognized. For example, if you code A,B,C as a dummy parameter list, substitutions will occur properly. However, you cannot use the accumulator or the B and C registers within the macro. Because of the limited scope of dummy parameters, the use of these registers is not affected outside the macro definition.

Dummy parameters in a comment are not recognized. No substitution occurs for such parameters.

Dummy parameters may appear in a character string. However, the dummy parameter must be adjacent to an ampersand character (&) as explained later in this chapter.

Any machine instruction or applicable assembler directive can be included in the macro body. The distinguishing feature of macro prototype text is that parts of it can be made variable by placing substitutable dummy parameters in instruction fields. These dummy parameters are the same as the symbols in the operand field of the MACRO directive.

Example:

Define macro MAC1 with dummy parameters G1, G2, and G3.

NOTE

The following macro definition contains a potential error that is clarified in the description of the LOCAL directive later in this chapter.

```
MAC1    MACRO   G1,G2,G3    ;MACRO DIRECTIVE
MOVES:  LHLD    G1          ;MACRO BODY
        MOV     A,M
        LHLD    G2
        MOV     B,M
        LHLD    G3
        MOV     C,M
        ENDM                ;ENDM DIRECTIVE
```

ENDM Directive

Label	*Opcode*	*Operand*
---	ENDM	---

The ENDM directive is required to terminate a macro definition and follows the last prototype instruction. It is also required to terminate code repetition blocks defined by the REPT, IRP, and IRPC directives.

Any data appearing in the label or operand fields of an ENDM directive causes an error.

NOTE

Because nested macro calls are not expanded during macro definition, the ENDM directive to close an outer macro cannot be contained in the expansion of an inner, 'nested' macro call. (See 'Nested Macro Definitions' later in this chapter.)

LOCAL Directive

Label	*Opcode*	*Operand*
---	LOCAL	label name(s)

The specified label names are defined to have meaning only within the current macro expansion. Each time the macro is called and expanded, the assembler assigns each local symbol a unique symbol in the form ??nnnn.

The assembler assigns ??0001 to the first local symbol, ??0002 to the second, and so on. The most recent symbol name generated always indicates the total number of symbols created for all macro expansions. The assembler never duplicates these symbols. The user should avoid coding symbols in the form ??nnnn so that there will not be a conflict with these assembler-generated symbols.

Dummy parameters included in a macro call cannot be operands of a LOCAL directive. The scope of a dummy parameter is always local to its own macro definition.

Local symbols can be defined only within a macro definition. Any number of LOCAL directives may appear in a macro definition, but they must all follow the macro call and must precede the first line of prototype code.

A LOCAL directive appearing outside a macro definition causes an error. Also, a name appearing in the label field of a LOCAL directive causes an error.

Example:

The definition of MAC1 (used as an example in the description of the MACRO directive) contains a potential error because the symbol MOVES has not been declared local. This is a potential error since no error occurs if MAC1 is called only once in the program, and the program itself does not use MOVES as a symbol. However, if MAC1 is called more than once, or if the program uses the symbol MOVES, MOVES is a multiply-defined symbol. This potential error is avoided by naming MOVES in the operand field of a LOCAL directive:

```
MAC1    MACRO   G1,G2,G3
        LOCAL   MOVES
MOVES:  LHLD    G1
        MOV     A,M
        LHLD    G2
        MOV     B,M
        LHLD    G3
        MOV     C,M
        ENDM
```

Assume that MAC1 is the only macro in the program and that it is called twice. The first time MAC1 is expanded, MOVES is replaced with the symbol ??0001; the second time, MOVES is replaced with ??0002. Because the assembler encounters only these special replacement symbols, the program may contain the symbol MOVES without causing a multiple definition.

REPT Directive

Label	*Opcode*	*Operand*
optional:	REPT	expression

The REPT directive causes a sequence of source code lines to be repeated 'expression' times. All lines appearing between the REPT directive and a subsequent ENDM directive constitute the block to be repeated.

When 'expression' contains symbolic names, the assembler must encounter the definition of the symbol prior to encountering the expression.

The insertion of repeat blocks is performed in-line when the assembler encounters the REPT directive. No explicit call is required to cause the code insertion since the definition is an implied call for expansion.