Z80 Assembler

Publication Number 87000068
Release
June 1981
$10.00

9520 SOFTWARE DEVELOPMENT SYSTEM

Z80 CROSS-ASSEMBLER SOFTWARE USERS MANUAL

Millennium Systems, Inc.
19050 Pruneridge Avenue
Cupertino, CA 95014
Telephone: (408) 996-9109
TWX/TELEX # 910-338-0256

Copyright @ 1981. No part of this publication may be reproduced without written permission from Millennium Systems, a subsidiary of American Microsystems, Inc.

PREFACE

This manual is intended to provide the user of a Millennium Systems 9520 Software Development System with sufficient knowledge to assemble and generate object code programs using the Millennium Systems Z80 Cross-Assembler. It should be understood that the information contained in this manual is valid only when the 9520 Software Development System is used to assemble programs that will be executed in Millennium Systems' 9508 MicroSystem Emulator unit.

The material in this manual is up-to-date at the time of publication, but is subject to change without notice.

Copies of this publication and other Millennium publications may be obtained from the Millennium sales office or distributor servicing your locality.

RELATED PUBLICATIONS

Other support documentation to be used in conjunction with this manual is as follows:

- Millennium Systems 9520 Software Development System Users Manual
- Millennium Systems 9508 MicroSystem Emulator Users Manual
- CP/M® 2.2 Interface Guide
- ED: A Context Editor for the CP/M Disk System Users Manual
- CP/M 2.2 Alteration Guide
- CP/M Dynamic Debugging Tool (DDT) Users Guide
- CP/M Assembler (ASM) Users Guide
- An Introduction to CP/M Features and Facilities
- CP/M 2.2 Users Guide
- MP/M® Multi-Programming Monitor Control Program Users Guide
- Word Star® Users Guide
- Applicable Addendums for 9508 MicroSystem Emulator Manual which describe emulation procedures for the following types of microprocessors:

  Z80
  8048/49/21/41
  6800/01/02/03
  8080/8085

ASSISTANCE

If you require any assistance on this product, please call Millennium Systems Customer Service on the toll-free, hot-line numbers listed below:

National	(800) 538-9320/9321
California	(800) 662-9231

CP/M® and MP/M® are trademarks of Digital Research.

WordStar® is a trademark of MicroPro International Corporation.

CONTENTS

Chapter Page

1 INTRODUCTION. 1-1

OVERVIEW. . . . .

ASSEMBLER INPUT . . . .

ASSEMBLER OUTPUT. . . . .

2 ASSEMBLER SOURCE MODULE FORMAT

INTRODUCTION. . . . .

Z80 SYMBOLIC STATEMENT FORMAT . . . .

THE LABEL FIELD . . . .

THE OPERATION FIELD . . . .

THE OPERAND FIELD . . . .

THE COMMENT FIELD . . . .

USING SYMBOLS . . . .

Programmer-Defined Symbols. . . . .

Pre-defined Symbols . . . .

Rules for Creating Symbols. . . . .

NUMERIC VALUES. . . . .

Scalar Values . . . .

Address Values. . . . .

NOTATION RULES FOR SPECIFYING CONSTANTS . . . .

Numeric Constants . . . .

String Constants. . . . .

Null Strings. . . . .

String to Numeric Conversion. . . . .

EXPRESSIONS PERMITTED IN THE OPERAND FIELD. . . . . . .

Hierarchy of Expression Operators and Functions . .

Description of Expression Operators and Functions .

Binary Arithmetic Operators . . . .

Unary Operators . . . .

Relational Operators. . . . .

Numeric Comparisons . . . .

String Comparisons. . . . .

String Concatenation. . . . .

Functions . . . .

STRING VARIABLES. . . . .

ASET Strings. . . . .

String Text Substitution. . . . .

3 STATEMENT SYNTAX CONVENTIONS

INTRODUCTION. . . . .

MILLENNIUM SYSTEMS ASSEMBLER STATEMENT SYNTAX . . . . .

Use of Upper and Lower Case Letters and Punctuation

Blank Fields. . . . .

Braces and Brackets . . . .

Trailing Dots . . . .

MP/M OR CP/M STATEMENT SYNTAX . . . .

Command Name. . . . .

Delimiters. . . . .

Parameters. . . . .

Trailing Dots . . . .

CONTENTS

Chapter | Page

4 ASSEMBLER DIRECTIVES
- INTRODUCTION
- LISTING FORMAT CONTROL DIRECTIVES
  - LIST and NOLIST
    - General Listing Format Control Options
    - Macro Listing Format Control Options
    - Conventions for Listing Control
- PAGE
- SPACE
- TITLE
- STITLE
- WARNING

SYMBOL DEFINITION DIRECTIVES
- EQU
- STRING
- ASET

LOCATION COUNTER CONTROL DIRECTIVE
- ORG

DATA STORAGE CONTROL DIRECTIVES
- BYTE
- WORD
- ASCII

MACRO DEFINITION DIRECTIVES
- MACRO
- ENDM
- REPEAT and ENDR
- INCLUDE

CONDITIONAL ASSEMBLY DIRECTIVES
- IF, ELSE, and ENDIF
- EXITM

SECTION DEFINITION DIRECTIVES
- Relocation Options
- SECTION
- COMMON
- RESERVE
- RESUME
- GLOBAL 4.50
- NAME 4.52

MODULE TERMINATION DIRECTIVE
- END

5 MACROS
- INTRODUCTION
- BASIC MACRO EXPANSION PROCESS
- MACRO DEFINITION DIRECTIVE
  - Macro Definition Directive Conventions

CONTENTS

Chapter Page

5 MACROS (cont.)

MACRO DEFINITION BLOCK. . . . .
Source Code Alteration. . . . .
Addditional Special Macro Definition Conventions. . . . . .
The @ Character . . . .
The # Character . . . .
The % Character . . . .
The ↑ or ∧ Character. . . . .
MACRO TERMINATION . . . .
MACRO CALLING . . . .
INCLUDE Directive Text Insertion. . . . .
Text Substitution . . . .
Special Macro Calling Characters. . . . .
The [] Construct. . . . .
The ↑ or ∧ Character. . . . .
Additional Macro Argument Conventions . . . .
EXAMPLES. . . . .
CONDITIONAL ASSEMBLY. . . . .
Nesting . . . .
Conditional Macro Termination . . . .
EXAMPLES. . . . .
IF-ENDIF Blocks . . . .
REPEAT-ENDR Blocks. . . . .
MACRO EXPANSION SUMMARY . . . .

6 ASSEMBLER OPERATING PROCESURES
INTRODUCTION. . . . .
PURPOSE . . . .
EXPLANATION . . . .

7 ASSEMBLER LISTING FORMAT
INTRODUCTION. . . . .
THE ASSEMBLER LISTING . . . .
Headings. . . . .
The Listing Line. . . . .
THE SYMBOL TABLE. . . . .

Appendices

A SOURCE MODULE CHARACTER SET . . . .
B ASSEMBLER DIRECTIVES. . . . .
C HEXADECIMAL CONVERSION TABLES . . . .
D ASSEMBLER ERROR CODES . . . .
E RESERVED WORDS. . . . .

Figure		Page
2-1	Properly Formatted Z80 Source Program . . . .
5-1	The Macro Expansion Process . . . .
7-1	Sample Assembler and Symbol Table Listing . . . .

Table	
2-1	Hierarchy of Expression Operators and Functions . . . .

INTRODUCTION

OVERVIEW

A Cross Assembler is an assembler program. It executes on one type of microprocessor-based system, and translates assembly language source programs into object modules for execution (after suitable link operation) on a different type of microprocessor-based system.

The Z80 Cross-Assembler executes on the 9520 Software Development System to assemble Z80 Assembly Language source programs into relocatable object modules. These object modules are linked appropriately (using the 9520 Linker Utility) to create load modules.

The load modules thus created, however, are executable only by downloading the object code from the 9520 system to a 9508 MicroSystem Emulator Unit which is configured with the Z80 Emulator Option components for execution. The load module can also be downloaded to a PROM Programmer where the user can burn a PROM that can be installed in, and executed at the 9508 MicroSystem Emulator.

The manual describes the Z80 Cross-Assembler as follows:

Chapter 1, Introduction and Overview, provides the user with an overview of the Z80 Cross-Assembler and basic information relative to the input/output operations of the assembler.

Chapter 2, Assembler Source Module Format, describes the format conventions which must be adhered to when using the assembler.

Chapter 3, Statement Syntax Conventions, explains and illustrates the syntax conventions used in the assembler.

Chapter 4, Assembler Directives, describes the assembler directives used in the MILLENNIUM SYSTEMS Assembler.

Chapter 5, Macros, explains the operation of macros.

Chapter 6, Assembler Operating Procedures, describes the syntax required to translate source code into executable binary object code.

Chapter 7, Assembler Listing Format, illustrates and explains the various parts of an assembler listing.

Assembler Input

The MILLENNIUM SYSTEMS Z80 Assembler translates user-written programs into executable binary format. The user's program must be written in Z80 symbolic notation (assembly language), and becomes the source module for assembler operation. User-written programs can be entered into disk files with the text editor program, using procedures described in the 9520 Software Development System Users Manual. If the source module is contained in more than one flexible disk file, each file name must be specified by assemble command (ASM) parameters.

All valid input devices can originate assembler input. The assembler reads the source module twice, once for each pass. When it encounters an END directive or reads the end of the last file during the first pass, the assembler begins the second pass and starts assembly.

Assembler Output

Assembler output comprises an object module, program listings, and appropriate information messages. The object module contains executable binary instructions and data constants translated from the source module. The entire object file must be linked and then loaded into program memory in order to execute the translated user program on the Z80 Emulator Processor.

Program listings produced by the assembler are composed of line numbers, the generated object code, and the source code as entered in the source module. Wherever an error is detected, an error code is printed on the display device and the user must refer to the listing to specify the nature of the problem.

Following the source code listing, a symbol table alphabetically lists all symbols entered in the program. The table also gives the hexadecimal value of each symbol and indicates undefined symbols. Below the symbol table, a message indicates the number of source lines, the number of assembled lines, the number of bytes available, and the number of errors and undefined symbols.

To transfer the listing and object file to a disk, enter output file names as ASM command parameters. To transfer assembler listing and object files to an output device (such as a line printer) instead of a file, specify the name of the device as the ASM command parameter.

The MILLENNIUM SYSTEMS Assembler makes two passes through the source module. The first pass determines the number of storage bytes required for each statement, and assigns a starting address value for the first byte of each statement line. The location counter, set to zero before the first pass begins, advances after each statement is read. This action effectively generates the starting address for each statement. The symbol table is also constructed during the first pass. During the second pass, the source module and the symbol table are used to generate the object module and the listings.

After assembly completion, each line containing an error is output to the display device, with an error code specifying the nature of the error. Below all error displays, a message indicates the number of source lines, the number of assembled lines, the number of bytes available, and the number of any errors or undefined symbols. If an irrecoverable error prevents assembly completion, the program aborts and an error code indicates the cause.

ASSEMBLER SOURCE MODULE FORMAT

INTRODUCTION

Symbolic Z80 instructions, assembler directives, macro calls, and explanatory comments form the source module. Each Z80 source module statement must be entered according to the MILLENNIUM SYSTEMS Z80 Assembler format. When translated by the assembler, the source module becomes the object module to be executed.

Three types of source module statements may be used:

1. Z80 symbolic instructions,
2. assembler directives, and
3. macro calls.

Z80 SYMBOLIC STATEMENT FORMAT

Each source module line may contain up to 128 characters, and is terminated by a carriage return. Allowable source module characters are detailed in Appendix A. Blank lines can be used to improve readability of the source module listing. The blank lines do not affect the translated program.

Each Z80 instruction, assembler directive, or macro call consists of four fields: the label field, the operation field, the operand field, and the comment field. During program assembly, each Z80 source module instruction is translated by the assembler into one, two, three, or four bytes of code in the object module. The length depends upon the instruction type, and the number and type of operands required.

The label field, when used, must begin in the first-character position of a line. The operation and operand fields must begin anywhere after the first-character position and end in any line character position within the 128-character range. The comment field may begin in any line character position and must end within the 128-character range. Field sequence may not be changed, however; and the correct order can only be as follows:

```
        LABEL        OPERATION        OPERAND        COMMENT
```

Throughout this manual, this field sequencing format is shown above each source line to illustrate proper assembler source line formatting.

Readability is improved when each field in the source module begins at a constant position within the line. This columnar forman can be easily implemented by using the tab setting function to define field starting positions. Figure 2-1 is an example of a properly formatted source module.

LABEL	OPERATION	OPERAND	COMMENT
	STRING	S1(80)	;DEFINE STRING VARIABLE S1 WITH ;80 CHARACTER MAXIMUM
L1	EQU	3	;DEFINE CONSTANT SYMBOL L1 TO EQUAL 3
L2	ASET	4	;DEFINE VARIABLE SYMBOL L2 TO EQUAL 4
	ORG	100H	;STARTS OBJECT CODE OF NEXT INSTRUC- ;TION AT 100H
	LD	A,(HL)	;LOAD REG.A WITH CONTENTS OF MEMORY ;POINTED TO BY HL REGISTER PAIR.
	END		;END OF PROGRAM

Figure 2-1. Properly Formatted Z80 Source Program

A general description of the characteristics of each source module field follows. MILLENNIUM SYSTEMS Assembler directives are described in Chapter 4 and listed in Appendix B. Macro calls are described in Chapter 5.

The Label Field

Labels may be used in all Z80 instructions, macro calls, and assembler directives. Every label must be unique within each source module. Duplicate labels prevent proper program execution and cause an error code to appear on the display device and in the listing. The label field, when used, must start in the first-character position of the line. A blank or tab terminates the label field; therefore, imbedded blanks or tabs are not permitted within the field.

Labels represent addresses associated with locations in a source module. The EQU and ASET directives are the only statements requiring label usage. In all other directives, label usage is optional. EQU and ASET directives always equate the required label to the constant or expression value in the operand field. The ASET directive allows the assigned symbol value to be modified; the EQU directive does not. For all other directives, the label meaning is dependent upon the particular directive. Generally, the label translates to the memory address of data or a data constant value. A label in a Z80 instruction translates to the address of the first byte of the instruction.

ORG and BLOCK directives must contain constants or operand symbols that have already been defined. Operands in all other directives may reference label symbols that are defined in later statements.

The Operation Field

The operation field contains the mnemonic operation code for a Z80 symbolic instruction, an assembler directive, or a macro call. The mnemonic specifies the operation or function to be performed at program execution time, or by the assembler during program translation and assembly. An instruction specifies the object code to be generated and the action to be performed on any operands that follow. An assembler directive specifies certain actions to be performed during assembly and might not generate any object code. The macro call specifies the macro definition block to be expanded.

The operation field begins after the label field is terminated. If the label is omitted, the operation field may begin anywhere after the first-character position in the line. The operation field is terminated by one or more spaces, by a tab or carriage return, or by a semicolon indicating the start of a comment field.

If the operation field does not contain a Z80 instruction, an assembler directive, or a macro call, the assembler rejects the entire statement and prints an error code. Four bytes of zero value are generated by the assembler to fill the area where a valid instruction would otherwise have been stored.

The Operand Field

The operand field specifies values or locations required for the given assembler directive, instruction, or macro call. The operand field, if present, begins after the operation field is terminated. Spaces may be used in the operand field. Two or more operands are separated by commas. The field is terminated by as carriage return, or by a semicolon indicating the start of a comment field.

The operation code (appearing in the operation field) determines the type and number of items required for the operand field. If more than one item is required, the sequence of item appearance is determined by the operation code.

Operands required for macro calls and assembler directives are discussed in Chapters 4 and 5, and summarized in Appendix B.

Nine types of information are permitted in the instruction operand field. Each instruction determines the operand types and their proper sequence. Refer to the Z80 Emulator Addendum for a summary of Z80 instruction requirements.

The following list defines the nine operand item types and their required syntax for Z80 instructions:

OPERAND ITEM TYPE	OPERAND ITEM SYNTAX
1) A Z80 register containing the operand data	A (8 bits) B (8 bits) C (8 bits) D (8 bits) E (8 bits) H (8 bits) I (8 bits) L (8 bits) R (8 bits) IX (16 bits) IY (16 bits) SP (16 bits)
2) A Z80 16-bit register pair containing the operand data	BC DE HL AF
*3) A 16-bit register pair enclosed within parentheses indicating a register holding an absolute memory address.	(BC) (DE) (HL)
4) An indexed expression indicating a memory address	(IX + expression)(IX - expression) (IY + expression)(IY - expression)
5) An 8-bit data or address constant within the range, -127 to +255. An immediate value.	Expression
6) A 16-bit data or address constant within the range, -32,768 to 32,767. An immediate value.	Expression
7) An 8-bit I/O device address within the range, 0 to 255.	(Expression)
8) A 16-bit operand data value.	Expression
9) A parenthesized 16-bit expression indicating a memory address.	(Expression)

*An expression which is only partially enclosed within parentheses is never a memory contents reference. Example:

```
LD A,(5+4); place contents of 9 into A
LD A,(5)+4; place the value 9 into A
```

The $ is used within operands to symbolize the first byte of the statement in which it appears. The effect of $ usage is equivalent to using a label in that statement. When using the $ to reference addresses, consult the Z80 Emulator Addendum for the number of bytes in each instruction. The two instruction sequences that follow are equivalent.

LABEL	OPERATION	OPERAND	COMMENT
1) TIMER	DEC	C	;DECREMENT C REGISTER, LABEL ;INSTRUCTION TIMER
	JR	NZ,TIMER	;JUMP BACK IF C NON-ZERO
2)	DEC	C	;DECREMENT C REGISTER
	JR	NZ,$-1	;JUMP BACK IF C NON-ZERO

The $ represents the address of the first byte in the JR instruction. Since the DEC instruction takes one byte, $-1 represents the first byte in the preceding instruction.

Caution should be exercised when using the $ symbol, since program logic errors could result. In the preceding example, an error might occur if an instruction were inserted between the DEC and JR instructions without changing the $-1 expression. Inserting an instruction in the first example requires no other changes.

Any symbols for the Z80 registers, and register pairs have been pre-defined by the assembler. Any data constant, or I/O device address in the operand field may be represented by expressions. An expression may consist of the following:

1) a single number,

2) a string constant,

3) a symbol, or

4) multiple numbers, string constants, and/or symbols combined with arithmetic and/or logical operations.

The assembler evaluates an expression in the operand field of a statement. If the expression violates permissible limits for the operand field, an error code is displayed. Additional information concerning expressions appears later in this section.

Any symbol appearing in the operand field that is not pre-defined by the assembler (see Pre-defined Symbols in this section) must be defined in the label field of an EQU or ASET directive or any Z80 instruction in the source module, or in the operand field of a GLOBAL, STRING, SECTION, COMMON, or RESERVE directive.

A statement may contain both the operand symbol and its label definition, as in the case of an instruction that jumps to itself. For example:

LABEL	OPERATION	OPERAND	COMMENT
HERE	JR	NZ,HERE	;HANG HERE IF PREVIOUS RESULT ;IS NON-ZERO

Typically, however, the symbol is defined in another statement. If the symbol is not defined in any statement, an error code is displayed. Additionally, symbols appearing in the operand field of ASET, EQU, ORG, and BLOCK directives must have been defined in the label field of a previous statement. Operand symbols in all other statements may be defined in the label fields of later statements.

If an illegal item appears in the operand field, the assembler flags the item with an error code on the display device and in the listing. All operand expressions are processed by the assembler to obtain 16-bit results. The assembler ignores any overflow conditions that occur while evaluating expressions. If the operand expression requires an 8-bit value and the value represented is greater than this, an error code is displayed and the assembler processes only the lower eight bits of the 16-bit value. An undefined value in the operand field is treated as zero, and causes an error.

The Comment Field

Programs containing comments are more readable, and hence easier to debug and modify. The optional comment field begins with a semicolon, is terminated by a carriage return, and follows all other statement fields. If no other fields are used, the comment field may begin anywhere in the statement.

String and macro substitution may be performed in the comment field. (Refer to the Chapter 2 subsection entitled String Text Substitution and to Chapter 5 for discussion on string and macro substitution.) Since the single quote character signals substitution, the character must be preceded by a caret (^) or up-arrow (↑) character when used for purposes other than substitution.

USING SYMBOLS

Symbol usage makes a program easier to read and modify, and reduces the risk of error during program modification. Symbols are defined when they appear in the label field of Z80 instructions, macro calls, and assembler directives, or in the operand field of GLOBAL, SECTION, COMMON, RESERVE, MACRO, or STRING directives. After having been defined, symbols can be used in the operation and operand fields of Z80 instructions, macro calls, and assembler directives.

A symbol label in a Z80 instruction represents the address of the first byte of that instruction. Such a label allows the user to transfer control (jump or call) to an instruction without knowing its absolute address. To transfer control, place the instruction symbol in the operand field of the jump or call instruction.

The meaning of a label symbol used as an operand for an assembler directive is dependent upon the directive. Generally, the symbol represents the memory address of data or a data constant value. Through the use of symbols, the directive operand field can refer to a data constant or a memory data area without regard to the absolute memory address. This is especially helpful when modifying a data constant frequently referred to by other statements. The programmer need only change the defining statement, rather than all statements referencing the constant.

Some symbols are created by the programmer, and others are pre-defined by the assembler.

Programmer-Defined Symbols

Programmer-defined symbols are assigned values during the assembler's first pass. Operand fields referring to the symbols are translated during the assembler's second pass. The ORG and BLOCK directives each alter the contents of the assembler location counter during both assembler passes. Because the alteration value is specified in the operand field of the ORG and BLOCK directives, any symbol appearing in the operand field of these directives must also be defined in the label field of a previous statement in the source module. The EQU directive operand field may contain a forward reference to a symbol, if the symbol does not appear in the operand field of an ORG, BLOCK, or another EQU directive. Forward referencing operand symbols are, however, allowed in all other statements.

Redefinition of symbols is generally not allowed. A previously defined ASET symbol, however, may be redefined in another ASET directive.

Pre-defined Symbols

Certain words are reserved as pre-defined symbol names for use in the operation and operand fields of source programs. Among these words are the following register symbols, assembler directives, instruction mnemonics, assembler listing options and operators. Refer to Appendix E for a complete list of reserved words for the Z80 Assembler.

1) The contents of 8-bit registers are specified by the character corresponding to the register name. The register names are A, B, C, D, E, H, I, L, and R.

2) The contents of 16-bit double registers and register pairs consisting of two 8-bit registers are specified by the two characters corresponding to the register name or register pair. The double register names are IX, IY, and SP. The register pair names are AF, BC, DE, and HL.

3) The Z80 instruction mnemonics (refer to Appendix E).

4) The Assembler directives, options, and operators (refer to Appendices B and E).

5) The MILLENNIUM SYSTEMS Assembler directives reserved for future use (refer to Appendix E).

Rules for Creating Symbols

The first character in a symbol must be alphabetic. The remainder of the symbol may be composed of the following characters: the letters A through Z; the numbers 0 through 9; and the special characters, . (period), _ (underscore), and $ (dollar sign). Lower-case letters are interpreted in their upper-case form. A symbol may contain up to eight characters. Only the first eight characters of the symbol are used, and excess characters are ignored. All pre-defined symbols are reserved words and cannot be redefined.

NUMERIC VALUES

The assembler defines two types of numeric values, scalars and addresses. Scalar values represent arbitrary numeric values. Address values represent actual memory locations within a program.

Scalar Values

Scalar values are signed integers ranging from -32,768 to +32,767. Scalar values serve as counting values in a program, rather than as actual references to memory locations. Scalar values are completely defined upon assembly.

Address Values

Address values represent actual memory locations within a user program. Address values are unsigned numbers ranging from 0 to 65,535. The assembler produces relocatable object code, that is, object code whose locations are defined during linking. Upon assembly, address values are relative to an assembler-defined base (or starting point). Therefore, actual memory locations associated with address values are unknown until after the linking process occurs.

More than one address base may exist within a given assembly. The user may define additional address bases by issuing a SECTION, COMMON, or RESERVE directive. Refer to Chapter 4 describing these directives and their relocation options. Since an address value lacks complete definition upon assembly, address value usage is more restrictive than scalar value usage. A unique location counter exists for each assembler-defined base in a user program. The $ symbol (current location counter contents) represents an address value.

NOTATION RULES FOR SPECIFYING CONSTANTS

Constants may be either numeric or string constants.

Numeric Constants

Numbers are integers and are assumed to be decimal unless otherwise specified. This means a number without a suffix is evaluated according to the decimal number base. A suffix letter code must be used to specify a radix other than decimal. The following suffixes are available:

1) H for hexadecimal. For example: 35H
   All numbers must begin with a numeric digit; therefore, a zero must precede all hexadecimal numbers beginning with the hexadecimal digits A through F. Examples of this follow:

   0B5H and 0FFH

2) O (capital o, not zero) or Q for octal. For example: 76O and 76Q

3) B for binary. For example: 10110110B

Leading zeros are appended to or truncated from constants to produce 8- or 16-bit values as required by the particular operand. Blanks are not permitted within a numeric constant. Refer to Appendix C for hexadecimal, decimal, and binary number conversion tables.

String Constants

In addition to symbols and numeric constants, operations may also contain string constants. String constants can be generated by using ASCII strings. ASCII (American Standard Code for Information Interchange) is a standard code for representing characters transmitted between the computer and peripheral devices such as teletypes, printers, and terminals. String constants and variables may be combined into string expressions using special operators. A string expression may be used anywhere a normal expression is allowed. String constants are written by enclosing ASCII characters within double quotes ("). A string constant may contain any character within the source code character set except a carriage return.

A double character may be included within a string by preceding it with a caret character (^). The caret character removes the special meaning from any character and allows the special character to be treated as a regular part of the text. A caret may also be included within a string by entering two carets. Examples of string constants and caret usage follow:

"ABCDEF"	results in the string	ABCDEF
"123^"34"	results in the string	123"34
"^^"	results in the string	^

Null Strings

A string containing zero characters is a null string. A null string is entered as two double quotes without intervening text or spaces ("").

String To Numeric Conversion

If a string expression is used where a numeric value is required, the string is automatically converted to a numeric value. The numeric value of a string is defined as follows:

> The numeric value of the null string ("") is zero.
>
> The numeric value of a one-character string is a 16-bit value whose high order nine bits are zeros and whose low order seven bits contain the ASCII code for the character.

The numeric value of a two-character string is a 16-bit value as well. In this case, the ASCII code for the leftmost character is in the high-order byte. The ASCII code for the second character from the left is in the low-order byte.

The numeric value of a string longer than two characters is the numeric value of the leftmost two characters in the string. An error code is displayed when this occurs.

Examples of string to numeric conversion follow. The numeric values for ASCII characters are found in Appendix C.

STRING	NUMERIC VALUE
""	0
"A"	41H
"12"	3132H
"123"	3132H (truncation error occurs)

EXPRESSIONS PERMITTED IN THE OPERAND FIELD

The operation field may contain an expression consisting of one or more terms acted on by expression operators. A term is either a symbol, a numeric constant, or an expression enclosed within parentheses. The value of a term may be an address, a scalar value, or undefined. Spaces are permitted within an expression; the assembler reduces the expression to a single value. When an invalid term is used, the display device and the listing show an error code, and the value of the expression is undefined.

The following outline lists the expression operators and functions. A chart describing the hierarchy of all expression operators and functions follows this summary. Each expression operator and function is described in greater detail, completing this discussion.

Unary Arithmetic Operators

OPERATOR	MEANING
+	identity
-	sign inversion

Binary Arithmetic Operators

OPERATOR	MEANING
*	multiplication
/	division
+	addition
-	subtraction
MOD	remainder
SHL	shift left
SHR	shift right

Unary Logical Operator

OPERATOR	MEANING
	not (bit inversion)

Relational Operators

OPERATOR	MEANING
=	equal
<>	not equal
>	greater than
<=	less than or equal
<	less than
>=	greater than or equal

Binary Logical Operators

OPERATOR	MEANING
&	and
!	inclusive or
!!	exclusive or

String Concatenation Operators

OPERATOR	MEANING
:	string concatenation

FUNCTIONS

HI (exp)

Returns the most significant byte of a numeric expression. The expression may be either an address or a scalar value. If an address is specified as the HI function argument, subsequent operations must not be performed on the HI function result. The HI function result is numeric.

LO (exp)

Returns the least significant byte of a numeric expression. The expression may be either an address or a scalar value. If an address is specified as the LO function argument, subsequent operations must not be performed on the LO function result. The LO function result is ~~a stringe~~ numeric.

DEF (sym)

Returns -1 (true) if the symbol has been previously defined in this pass. Otherwise, returns 0 (false). The DEF function result is numeric.

SEG (string expression,exp1,exp2)

Extracts exp2 characters from the specified string, starting with the character, exp1. If the end of the string is encountered before exp2 characters are extracted, only those characters up to the string end are extracted. Both exp1 and exp2 must be scalar values. The SEG function result is a string.

NCHR (string expression)

Returns the current number of characters in the specified string. For a string variable, the length returned may be less than the length defined by the STRING directive. The NCHR function result is numeric.

ENDOF (section name)

Upon linking, the ENDOF function returns the address of the last byte of the specified section. The symbol specified in this function must be a global symbol. If the symbol is not a section name, the address of the symbol is returned. Further operations may be performed on the result of ENDOF, provided the operations are allowed for address values. The ENDOF function result is numeric.

BASE (exp1,exp2)

Returns -1 (true) if the two expressions, exp1 and exp2, share the same base. Otherwise, returns 0 (false). The BASE function result is numeric.

STRING (exp)

Returns the value of the expression as a six-character string. The five rightmost characters represent the decimal value of the expression; the leftmost character indicates whether the number is positive or negative. If the leftmost character is a minus, "-", the number is negative. If that character is a zero, "0", the number is positive. The expression must be a scalar value.

SCALAR (exp)

Converts the address value of the expression to a scalar value.

Hierarchy of Expression Operators and Functions

In multiple-operator expressions, operators and functions are evaluated in the order of their precedence. Table 2-1 illustrates this hierarchy. The functions at the top of the table have the highest precedence. The operators at the bottom of the table have the lowest precedence. All expression operators and functions located on the same line have equal precedence, and are evaluated from left to right. Parentheses may be used to override the order of precedence, and parentheses are evaluated from inward to outward. The most deeply parenthesized subexpressions are evaluated first.

If the expression entered is too complex for the assembler to translate, an expression error code is displayed. This does not occur when parentheses nesting depth is three or less.

Table 2-1. Hierarchy of Expression Operators and Functions.

LO	HI	SEG	NCHR	DEF	ENDOF	BASE	STRING	SCALAR
:								
+	- (unary plus and minus)							
*	/	SHL	SHR	MOD				
+	- (addition and subtraction)							
=	<>	<	<=	>	>=			
&								
!	!!							

Description of Expression Operators and Functions

In addition to the arithmetic (+, -, *, /) and logical (, &, !, !!) operators, several other operators and functions are allowed within numeric expressions. These operators and functions provide additional arithmetic functions and a means for comparing numeric quantities.

Binary Arithmetic Operators

Binary arithmetic operators act on numeric values, which may be scalar or address values. Scalar values may appear within arithmetic operations in any combination. Only the following binary arithmetic operations are permitted when acting upon addresses:

```
SCALAR VALUE + ADDRESS      = ADDRESS

ADDRESS      + SCALAR VALUE = ADDRESS

ADDRESS      - SCALAR VALUE = ADDRESS

ADDRESS      - ADDRESS      = SCALAR VALUE (Both addresses must be based
                                            to the same section.)
```

Any other combination of address terms yields an undefined result.

MOD is a binary operator that computes the remainder when the first operand is divided by the second operand. For example, an instruction entered as A MOD B yields the remainder resulting from A/B. The program segment that follows demonstrates MOD operator usage.

LABEL	OPERATION	OPERANDS	COMMENT
AX	EQU	5 MOD 2	;AX IS SET TO 1, SINCE 5/2 YIELDS A ;REMAINDER OF 1
BX	EQU	14 MOD AX	;BX IS SET TO 0, SINCE 14/1 YIELDS A ;REMAINDER OF 0
CZ	EQU	(BX+29)MOD 25	;CX IS SET TO 4, SINCE 0+29 YIELDS 29 ;AND 29/25 YIELDS A REMAINDER OF 4
DX	EQU	(-5) MOD 2	;DX IS SET TO -1, SINCE -5/2 YIELDS A ;REMAINDER OF -1

SHL and SHR are binary operators that shift their first operands the number of bit positions specified by their second operands.

SHL performs a left logical shift (equivalent to multiplying by two). Zeros are shifted into the right end of the 16-bit value. Bits shifted out of the leftmost bit position are lost.

SHR performs a right logical shift. Zeros are shifted into the leftmost bit positions. Bits shifted from the rightmost bit position are lost. Shifts of 16 or more bits generate a result of zero and produce a truncation error code. The program segment that follows demonstrates SHL and SHR operator usage.

LABEL	OPERATION	OPERAND	COMMENT
DX	EQU	1 SHL 1	;VALUE ASSIGNED TO DX IS 2, SINCE A ;SHIFT LEFT ONCE CAUSES 1 TO BE ;MULTIPLIED BY 2
EX	EQU	DX SHR 1	;VALUE ASSIGNED TO EX IS 1 SINCE DX ;(2) SHIFTED RIGHT IN A BINARY FASHION ;YIELDS 1
FX	EQU	06E0H SHL 3	;VALUE ASSIGNED TO FX IS 3700H, ;SINCE 2 CUBED IS 8, AND 8 TIMES ;06E0H is 3700H
GX	EQU	0FFFFH SHR 16	;VALUE ASSIGNED TO GX IS 0, SINCE ;0FFFFH SHIFTED RIGHT IN A BINARY ;FASHION YIELDS 0

Unary Operators

All unary operators may act upon scalar values. The plus sign (+) is the only unary operator permitted to act upon addresses.

Relational Operators

The relational operators include =, <>, >, <, <=, and >=. Relational operators allow signed numeric, unsigned numeric, and string comparisons.

Numeric Comparisons

If either of the operands of a relational operator is numeric, the relational operators perform signed or unsigned numeric comparisons. A signed numeric comparison is performed on two scalar values, a string and a scalar value, or a scalar and a string value. An unsigned numeric comparison is performed whenever one of the operands is an address. Comparison of two addresses based in different sections results in an undefined value. These comparisons are summarized as follows:

	STRING	SCALAR	ADDRESS
STRING	String Comparison	Signed Numeric Comparison	Unsigned Numeric Comparison
SCALAR	Signed Numeric Comparison	Signed Numeric Comparison	Unsigned Numeric Comparison
ADDRESS	Unsigned Numeric Comparison	Unsigned Numeric Comparison	Unsigned Numeric Comparison

If a comparison is performed between an address and a string or scalar value, the base of the address is first added to the string or scalar value. If two addresses are compared, they must have the same base, or an error results.

For signed comparisons, numbers range from -32768 to 32767. For unsigned comparisons, numbers range from 0 to 0FFFFH (65,535).

An operator in a numeric comparison determines whether the specified relationship exists between its two operands. The resulting value is 0 if the relationship is false and -1 (0FFFFH) if the relationship is true. Examples of relational operator usage follow.

LABEL	OPERATION	OPERAND	COMMENT
T	EQU	-5 > 7	;VALUE ASSIGNED TO T IS 0, SINCE -5 ;IS NOT GREATER THAN 7
P	EQU	7 > = -5	;VALUE ASSIGNED TO P IS -1, SINCE 7 ;IS GREATER THAN -5
U	EQU	T < > P	;VALUE ASSIGNED TO U IS -1, SINCE R ;IS NOT EQUAL TO P

String Comparisons

The relational operators (=, <>, >, <, <=, >=) may be used to compare the values of two string expressions. When strings are compared using these relational operators, the comparison is made numerically, according to the ASCII collating sequence. Refer to Appendix C for the correct character ordering sequence of ASCII characters.

String comparison is performed only when both operands of a relational operator are strings. If only one of the operands of a relational operator is a string, the string is converted to a scalar value and a numeric comparison is performed.

String comparison always proceeds from left to right. If two strings are equal through the last character of the shorter string, the shorter string is considered to be less than the longer string.

Examples of string comparisons follow.

"AB" = "AB" results in -1 (true)

"AB" < > "AB" results in 0 (false)

"A" > "B" results in 0, since A is less than B

"ABC" > "AAAA" results in -1, since B is greater than A

"ABC" > "ABC" results in 0, since "ABC" has three characters "ABC" has four, including the final space

""<"". results in -1, since a null string is less than a blank character

1 < "1" results in -1, since the numeric value of the ASCII character "1" is 31H and is greater than 1

String Concatenation

The concatenation operation combines two strings into a single string. The operator used to specify string concatenation is the colon (:). The colon may be used to concatenate any two string expressions. An error occurs when an attempt is made to concatenate two numeric values or a string and a numeric value. Examples of string concatenation follow:

"A":"B" results in "AB"

"":"" results in "", since two null strings produce a null string

"A":"":"B" results in "AB", since a null string and a character produce the character

"A":"" results in "A "

"ABC":"1":"2" results in "ABC12"

Functions

HI and LO are unary functions that respectively extract the high- and low-order eight bits of their operands. References to HI or LO are written as single argument functions. The value to be acted on appears in parentheses, following the keyword HI or LO. If this value is an address, further operations on the result of HI or LO are disallowed. Examples of HI and LO function usage follow:

LABEL	OPERATION	OPERAND	COMMENT
IXB	EQU	HI (0C00FH)	;VALUE ASSIGNED TO IXB IS C0H
JX	EQU	LO (0C00FH)	;VALUE ASSIGNED TO JX IS 0FH
KX	EQU	LO (HI(0C00FH)	;VALUE ASSIGNED TO KX IS C1H
Z	EQU	5 + LO(Q)	;INVALID WHEN Q IS AN ADDRESS

DEF is a unary function that determines whether a symbol has already been defined. DEF is referenced as a single-argument function. The argument must be a symbol and may not be an expression. If the argument symbol has already been defined, the value of DEF is -1 (0FFFFH). If the argument has not been defined, the value of DEF is 0. A pre-defined symbol used as an argument causes an error. Examples of DEF function usage follow.

LABEL	OPERATION	OPERAND	COMMENT
MK	EQU	DEF(K)	;VALUE ASSIGNED TO MK IS -1 IF K IS ;ALREADY DEFINED
Q	EQU	DEF(N)	;VALUE ASSIGNED TO Q IS 0 IF N IS ;UNDEFINED
RX	EQU	DEF(RX)	;VALUE ASSIGNED IS 0. THE SYMBOL ON ;THE LEFT OF THE EQU DIRECTIVE IS ;UNDEFINED UNTIL THE EXPRESSION ;ON THE RIGHT IS EVALUATED
S	WORD	DEF(S)	;A WORD OF OBJECT CODE CONTAINING ;0FFFFH(-1) IS GENERATED. THE LABEL ;ON THE WORD STATEMENT IS DEFINED ;BEFORE THE STATEMENT IS EVALUATED.

The SEG function (segmentation) is used to extract a portion of a string. The SEG function uses three arguments. The first argument is the string (or string expression) from which a substring is to be extracted. The second argument is a numeric expression specifying the position of the leftmost character of the string where the substring is to be extracted. Characters within the string are numbered from left to right starting with one. The third argument is a numeric expression specifying the number of characters to be extracted. The specified characters are extracted unless the end of the string is encountered first. In this case, only those characters up to the end of the string are extracted. The following examples illustrate properties of the SEG function:

```
SEG("ABCD",2,2)        results in     "BC"

SEG("ABCD",1,4)        results in     "ABCD"

SEG("ABCD",3,3)        results in     "CD"

SEG("ABCD",5,2)        results in     ""(the null string, resulting in
                                         zero characters)

SEG("ABCD",3,0)        results in     ""
```

The NCHR function may be used to determine the number of characters in a string expression. NCHR is referenced as a single-argument function, that argument being the string expression whose length is to be determined. The result of NCHR is numeric and not a string value. Examples of NCHR function results follow.

```
NCHR ("")                    results in     0

NCHR("ABC")                  results in     3

NCHR(SEG("XYZ",2,1)          results in     1

SEG("ABC",NCHR("ABC"),1)     results in     "C", since C is the last
                                                character of "ABC"
```

The ENDOF function returns the address of the last byte of a section. The argument for ENDOF must be the section name whose ending address is to be determined. An example of ENDOF usage follows:

```
LABEL     OPERATION     OPERAND            COMMENT

          .
          .
          .
          RESERVE       STACK,100H         ;NAMES A SECTION, STACK, AND
                                           ;ALLOCATES AT LEAST 256 BYTES

          LD            SP,ENDOF (STACK)   ;LOAD STACK REGISTER WITH THE END
                                           :OF THE STACK
```

The BASE function determines whether two expressions share the same base. If the expressions share the same base, the value of BASE is true (OFFFFH). Otherwise, the value of BASE is false (0). Examples of BASE function results follow. Q,R, and ZZ represent addresses where Q and R share a common base, while ZZ does not.

BASE (Q,R)	results in	OFFFFH (true)
BASE (Q,Q+15)	results in	OFFFFH (true)
BASE (ZZ,Q)	results in	0 (false)
BASE (Q,Q-R)	results in	0 (false) because Q-R is scalar
BASE (5,15)	results in	OFFFFH (true) because 5 and 15 are both scalar
BASE (5,Q-R)	results in	OFFFFH (true)
BASE (5,ZZ-Q)	results in	Error since subtraction is not valid between addresses with different bases

The STRING function returns the decimal value of an expression as a six-character string. The expression must be a scalar value. When the value does not fill six digits, leading zeros appear in the resulting string. If the expression value is negative, a minus sign is placed in the resulting string. Examples of STRING function results follow:

STRING(5)	results in	"000005"
STRING(5+15)	results in	"000020"
STRING(OFFH)	results in	"000255"
STRING(-OFFH)	results in	"-00255"

The SCALAR function converts the address value of the expression to a scalar value. The resulting scalar value is equal to the displacement of the address value from the address value's base. Upon linking, the resulting scalar value might not be the same as the final value of the expression. The SCALAR function does not affect scalar-valued expressions.

An example of scalar conversion follows:

LABEL	OPERATION	OPERAND	COMMENT
	SECTION	X	;DEFINES A NEW SECTION NAMED :X
A1	ORG	7	;ADVANCES LOCATION COUNTER ;TO ADDRESS 7. ASSIGNS ADDRESS ;7 TO A1
	WORD	SCALAR($) MOD 2	;CONVERTS ADDRESS 7 TO SCALAR ;VALUE. PERFORMS 7/2 AND ;RETAINS REMAINDER 1. ;ALLOCATES ONE WORD TO ;VALUE 1
	SECTION	ASDF	;DEFINES NEW SECTION NAMED ;ASDF
A2	ORG	6	;ADVANCES LOCATION ;COUNTER TO ADDRESS 6 WITHIN ;SECTION ASDF. ASSIGNS 6 TO A2
	WORD	SCALAR(A1)+SCALAR(A2)	;ALLOCATES ONE WORD ;CONTAINING SCALAR VALUE 13

Note that if the SCALAR function were not entered in the above WORD directives, an error would result. Scalar values are unaffected by changes in address base. Thus, in the above program, the scalar result of the operation WORD SCALAR(A1) + SCALAR(A2) remains unchanged no matter what base values are assigned to sections X and ASDF upon linking.

STRING VARIABLES

String variables enhance the value of string expressions by providing a means for storing string expression values. A string variable is a symbol with an associated string value, and is created by use of the STRING directive.

The desired string variable names are defined in the operand field of the STRING statement. The maximum character length of the value to be stored in the string variable may be specified by entering a numeric expression in the operand field. When this optional character length expression is not specified, an eight-character length is assumed. In the following example, a string variable is defined as STRVAR, with a maximum character length of 16.

LABEL	OPERATION	OPERAND
	STRING	STRVAR(16)

For further discussion pertaining to STRING statements, refer to Chapter 4 describing assembler directives.

ASET Strings

The ASET directive assigns a string expression value to a string variable defined with the STRING directive. The string variable is entered in the label field of the ASET directive; the string expression is entered in the operand field. The string expression value is evaluated and assigned to the string variable. If the resulting string expression's length is longer than the maximum string variable length, the string expression is truncated before assignment, and an error code is displayed. Examples of ASET string usage follow.

LABEL	OPERATION	OPERAND	COMMENT
	STRING	A1,A2(2),A3(45),A4(0)	;DEFINES STRING VARIABLE A1 ;WITH A DEFAULTING VALUE ;LIMIT OF 8 CHARACTERS ;DEFINES STRING VARIABLES ;A2, A3, AND A4 WITH ;RESPECTIVE VALUE LIMITS OF ;2, 45, AND 0 CHARACTERS
A1	ASET	"AB"	;VALUE OF A1 IS "AB"
A2	ASET	A1	;VALUE OF A2 IS "AB"
A4	ASET	A1:A2	;VALUE OF A4 IS "" ;TRUNCATION ERROR SINCE A4 ;ALLOWS A VALUE LIMIT OF 0 ;CHARACTERS
A3	ASET	"A MEDIUM LONG STRING"	;VALUE OF A3 is "A ;MEDIUM LONG STRING"
A1	ASET	A3	;VALUE OF A1 IS "A MEDIUM". ;STRING TRUNCATED

String Text Substitution

String variables may be used for modification of source text being processed by the assembler. Using string variables makes it possible to insert code into a source line, thus allowing the code to be processed as if it were part of the original source line. Before the assembler processes a source line, it scans the line for string variables enclosed within single quote characters. When such a variable is encountered, it is replaced with the specified value and the scan continues. When the entire line has been scanned and all code substitutions are made, the assembler then processes the line. For example assume the assembler processes the following code:

LABEL	OPERATION	OPERAND
	STRING	OP
OP	ASET	"WORD"
	'OP'	1,2,3

When the assembler scans the line containing 'OP' 1,2,3, the string variable 'OP' is replaced with the value defined for the substitution, "WORD". The line resulting upon assembly follows:

```
WORD 1,2,3
```

String substitutions can occur anywhere within a line of code including within string constants and comments. For the examples that follow, assume that A1, A2, A3, and A4 are defined as specified.

LABEL	OPERATION	OPERAND
	STRING	A1,A2,A3,A4
A1	ASET	"YTE"
A2	ASET	"123,456"
A3	ASET	"COMMENT"
A4	ASET	""

Assume that the following substitutions are then performed.

SOURCE CODE	RESULTS AFTER SUBSTITUTION
BYTE 'A1','A2'	BYTE YTE,123,456
WORD 1 'A4'	WORD 1
A4 ASET" 'A3' "	A4 ASET "COMMENT"
WORD " 'A4' "	WORD "COMMENT"
B'A1''A2'-200	BYTE 123,456-200
B'A1'-'A2'	BYTE 123,456 (error code displayed due to undefined instruction mnemonic, since space was omitted between 'A1' and 'A2')

Since the single quote character always signals string substitution, it is necessary to precede the character with a caret (^) if string replacement is not be be performed. The caret character allows the single quote character to then be interpreted as a literal character in a statement. An example demonstrating caret usage follows:

ASCII "WHAT 'S UP DOC?" results in WHAT'S UP DOC?

STATEMENT SYNTAX CONVENTIONS

INTRODUCTION

Many of the following chapters in this manual contain MILLENNIUM SYSTEMS Assembler and MP/M or CP/M statement descriptions. Each statement description is preceded by a syntactical block showing the required statement format. This section describes the syntax conventions for MILLENNIUM SYSTEMS Assembler and MP/M or CP/M statements.

MILLENNIUM SYSTEMS ASSEMBLER STATEMENT SYNTAX

MILLENNIUM SYSTEMS Assembler directives and macro calls may contain up to four fields. Each field name is indicated in the syntactical block above the corresponding field item, as shown in the following example.

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	BYTE	{expression} [,expression]	[;charstring]

Use of Upper and Lower Case Letters and Punctuation

A capitalized item in a field must be entered exactly as shown. Punctuation delimiters such as commas, semicolons, or parentheses must also be entered exactly as shown. Spaces or tab characters terminate each field and begin the next. An item shown in lower case letters is a term signifying the entry type. The following descriptive terms are used to signify entry type unless otherwise specified:

1) symbol - as defined in Chapter 2

2) expression - as defined in Chapter 2

3) charstring - a string of one or more characters.

Blank Fields

Any field left blank is an illegal field for that statement.

Braces and Brackets

When an item is enclosed in braces { }, the item must be present in the statement. Items enclosed in brackets [ ] are optional. Braces and brackets are used for syntactical representation only and should not be entered as part of the statement. Braces and brackets may be nested. The following is an example of braces and brackets nested in braces.

{{strvar 1} [lenexp 1]}

Trailing Dots

A line of dots following an item indicates that the item can be repeated a number of times. The item cannot be repeated beyond the end of the line being entered. In the example that follows, the item can be repeated.

[,symbol]...

MP/M OR CP/M STATEMENT SYNTAX

A MP/M or CP/M statement contains a command and in some cases, one or more parameters with delimiting characters. An example of a typical MP/M or CP/M statement syntactical block follows:

```
SYNTAX

        {COMMAND} [device I.D.] : {file name} [.file type]
```

Command Name

The command name (eight characters, maximum) identifies a MP/M or CP/M system utility name.

Delimiters

Items in the command line must be separated by delimiters when entered into the terminal. A space is used as the main delimiter. The colon is used to delimit the device identification and the file name. The period, which is necessary only when the file type is specified, is used to delimit the file name and file type.

Parameters

The parameters or controlling conditions of each command line are shown in the preceding MP/M or CP/M statement syntactical block. Each parameter may consist of a file name, a function, a device name, an indicator or an assigned value. When a parameter is shown capitalized, it must be entered exactly as shown. Parameters shown in lower case letters are descriptive terms to signify the type of entry.

Braces and Brackets

When appearing MP/M or CP/M statement, syntactical descriptions, braces and brackets have the same meaning as when used with MILLENNIUM SYSTEMS Assembler statements. Additionally, parameters stacked within either braces or brackets indicate that only one of the enclosed items should be selected for statement entry. In the following example, an object file name or an object device may be selected, but not both.

```
[ object file name ]
[ object device    ]
```

Trailing Dots

Trailing dots within MP/M or CP/M statement syntactical blocks indicate repetitive parameters.

ASSEMBLER DIRECTIVES

INTRODUCTION

The following assembler directives are available:

Listing Format Control Directives

LIST
NOLIST
PAGE
SPACE
TITLE
STITLE
WARNING

Symbol Definition Directives

EQU
STRING
ASET

Location Counter Control Directive

ORG

Data Storage Control Directives

BYTE
WORD
ASCII
BLOCK

Macro Definition Directives

MACRO
ENDM
REPEAT
ENDR
INCLUDE

Conditional Assembly Directives

IF
ELSE
ENDIF
EXITM

Relocatable Section Definition Directives

SECTION
COMMON
RESERVE
RESUME
GLOBAL
NAME

Module Termination Directive

END

LISTING FORMAT CONTROL DIRECTIVES

The assembler listing format directives are presented in the order shown below:

Mnemonic	Purpose
LIST	Enables display of assembler listing features.
NOLIST	Disables display of assembler listing features.
PAGE	Begins the next listing line on the following page.
SPACE	Spaces downward a specified number of listing lines.
TITLE	Creates a text line at the top of each listing page for program identification.
STITLE	Creates a text line on the second line of each listing page heading for program identification.
WARNING	Upon assembly, generates a warning message on the output device and in the listing. Also allows the user to specify his own warning message.

LIST/NOLIST

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	LIST	[CND] [,TRM] [,SYM] [,CON] [,MEG] [,ME]	[;charstring]
[symbol]	NOLIST	[CND] [,TRM] [,SYM] [,CON] [,MEG] [,ME]	[;charstring]

Purpose

Two assembler listing control directives, LIST and NOLIST, respectively enable and disable display of assembler listing features.

Explanation

When NOLIST is specified without operands, all output to the listing file (except the symbol table) is suppressed. When LIST is entered without operands, the listing is turned back on.

General Listing Format Control Options

Four general listing control options (CND, TRM, SYM, and CON) may be entered with the listing control directive, LIST, when specific features in the assembler listing are desired for viewing. The same four listing options may be entered with the assembler listing control directive, NOLIST, when specific features in the assembler listing are not desired for viewing.

The general listing control options are summarized as follows:

CND - Lists unsatisfied conditions for IF and REPEAT operations. (Refer to the subsections describing macro definition directives and conditional assembly directives.) The listing defaults to an OFF condition, thus displaying only those instructions within an IF or REPEAT condition occurring when the condition is satisfied.

TRM - Causes the listing to be trimmed to a 72-character format during display. Defaults to an OFF condition, causing the listing to be displayed in the standard 132-character format.

LIST/NOLIST (Continued)

The general listing control options are summarized as follows: (continued)

SYM - Lists the symbol table. Defaults to an ON condition.

CON - Displays all assembly errors to the console. Defaults to an ON condition.

Macro Listing Format Control Options

A macro is a shorthand approach for inserting a pre-defined source code block into a program. Refer to KChapter 5 for a discussion of macro procedures.

Only those macro instructions generating object code appear in an assembler listing. Some of the code generated during a macro expansion does not generate object code upon assembly, making it impossible under normal conditions to view the entire macro expansion sequence within the assembler listing. Therefore, in addition to the four general listing control options, two macro listing control options (MEG and ME) may be entered with the LIST and NOLIST directives to enable and disable macro expansion visibility. These options are summarized as follows:

MEG - Lists only macro expansion code that changes the location counter. Defaults to an ON condition.

ME - Lists all macro expansion code except for any unsatsified IF or REPEAT conditions. When the listing control option CND is on, unsatisfied conditions are also listed. Defaults to an OFF condition. If either ME or MEG is turned OFF by the user, the other is automatically turned OFF. If ME is turned ON by the user, MEG is automatically turned ON.

The following table demonstrates LIST AND NOLIST effects on the ME and MEG options:

ENTRY	RESULTS	
NOLIST MEG	MEG is OFF.	ME is OFF.
NOLIST ME	MEG is OFF.	ME is OFF.
LIST MEG	MEG is ON.	ME is OFF.
LIST ME	MEG is ON.	ME is ON.
NOLIST	MEG is OFF. Status of both options saved.	ME is OFF.
LIST	Restores status of both options.	

Upon exit from a macro expansion, the main program listing status is restored to the status that prevailed before the macro was called.

LIST/NOLIST (Continued)

Conventions for Listing Control

The LIST and NOLIST directives are always entered in the operation field of the listing control statements where they appear. More than one listing control option may be entered with the LIST and NOLIST directives. In this case, each option is separated from other options by a comma. When entering the listing control options with the LIST or NOLIST directives, the options are placed in the operand field of the listing control directive in any order. If the NOLIST directive is entered without options to suppress display, and the LIST directive is again entered without options specified, the original specified options are retained. The number on any listing line corresponds to the original input source line number. The NOLIST directive does not affect this line number correlation.

Examples

The following listing control statement suppresses the symbol table listing.

LABEL	OPERATION	OPERAND	COMMENT
	NOLIST	SYM	;SUPPRESSES SYMBOL TABLE LISTING

The following listing control statement causes all subsequent macro expansions and unsatisfied conditions to be included within the assembler listing.

LABEL	OPERATION	OPERAND	COMMENT
	LIST	ME,CND	;LISTS MACRO EXPANSIONS ;AND ALL UNSATISIFED ;CONDITIONS

PAGE

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	PAGE		[;charstring]

PAGE (Continued)

Purpose

The PAGE directive causes the next listing line to begin on the following page.

Explanation

As the source lines are read by the assembler in its second pass, they are output to the listing along with any object code produced. When the PAGE directive is encountered, a page heading is printed at the top of the new page and the next listing line begins below the heading. The actual PAGE directive is not printed in the listing.

A label is generally not used with the PAGE directive; however, if used, the symbol represents the address in the assembler location counter. The location counter contains the address of the next instruction or data byte in the program sequence.

LABEL	OPERATION	OPERAND	COMMENT
	STRING	S1 (80)	;DEFINE STRING VARIABLE S1 ;WITH 80-CHARACTER ;MAXIMUM
L1	EQU	3	;DEFINE CONSTANT SYMBOL ;L1 TO EQUAL 3
L2	ASET	4	;DEFINE VARIABLE SYMBOL L2 ;TO EQUAL 4
	PAGE		;BEGINS NEW LISTING PAGE
	ORG	100H	;STARTS OBJECT CODE OF NEXT ;INSTRUCTION AT 100H
	LD	A,(HL)	;LOADS THE CONTENTS OF ;MEMORY POINTED TO BY THE ;HL REGISTER PAIR INTO ;REG. A
	END		;END OF PROGRAM

PAGE (Continued)

Upon assembly, the following listing file results from this source program. A new page is generated after the ASET directive.

```
MILLENIUM Z80 ASM V3.3                                                      PAGE 1

00001                    STRING S1 (80)      ;DEFINE STRING VARIABLE S1
                                             ;WITH 80-CHARACTER
                                             ;MAXIMUM

00002       0003 L1      EQU    3            ;DEFINE CONSTANT SYMBOL
                                             ;L1 TO EQUAL 3

00003       0004 L2      ASET   4            ;DEFINE VARIABLE SYMBOL
                                             ;L2 TO EQUAL 4

MILLENIUM Z80 ASM V3.3                                                      PAGE 2

00005       0100 >       ORG    100H         ;STARTS OBJECT CODE OF
                                             ;NEXT INSTRUCTION AT 100H

00006  0100 7E           LD     A,(HL)       ;LOADS THE CONTENTS OF
                                             ;MEMORY POINTED TO BY THE
                                             ;HL REGISTER PAIR INTO
                                             ;REGISTER A

00007                    END                 ;END OF PROGRAM
.
.
.
MILLENNIUM Z80 ASM V3.3   SYMBOL TABLE LISTING                              PAGE 3

STRINGS AND MACROS

  S1 - - - - - 0050 S

SCALARS

  L2 - - - - - 0004 V

% (default) SECTION 0001

  L1 - - - - - (0101)

7 SOURCE LINES    7 ASSEMBLED LINES    1000 BYTES AVAILABLE
```

PAGE (Continued)

Note that the symbol indicators V and S respectively follow the symbols L2 and S1. The symbol indicator V indicates that L2 is an ASET symbol. The symbol indicator S indicates that S1 is a string. The symbol L1 has no symbol indicator following it, indicating that L1 is an EQU symbol. For a more complete description of symbol indicators, refer to Chapter 7, entitled ASSEMBLER LISTING FORMAT.

SPACE

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	SPACE	[expression]	[;charstring]

Purpose

Whenever the SPACE directive appears in the source module, the assembler spaces downward a specified number of lines in the listing.

Explanation

The number of lines to be spaced downward is indicated by the expression in the SPACE directive operand field. If no expression is entered, one space is generated. If the execution of the SPACE directive crosses a page boundary, the effect is the same as that of the PAGE directive. The actual SPACE directive is not printed in the listing.

A label is generally not used with the SPACE directive; however, if used, the symbol represents the address in the assembler location counter. The location counter contains the address of the next instruction or data byte in the program sequence.

SPACE (Continued)

Example

Assume the following source program resides on disk.

LABEL	OPERATION	OPERAND	COMMENT
	STRING	S1 (80)	;DEFINE STRING VARIABLE S1 ;WITH 80-CHARACTER MAXIMUM
L1	EQU	3	;DEFINE CONSTANT SYMBOL ;L1 TO EQUAL 3
L2	ASET	4	;DEFINE VARIABLE SYMBOL ;L2 TO EQUAL 4
	SPACE	10	;SPACES DOWNWARD 10 ;LISTING LINES
	ORG	100H	;STARTS OBJECT CODE OF ;NEXT INSTRUCTION AT 100H
	LD	A,(HL)	;LOADS THE CONTENTS OF ;MEMORY POINTED TO BY THE ;HL REGISTER PAIR INTO ;REG. A
	END		;END OF PROGRAM

SPACE (Continued)

Upon assembly, the following listing file results from this source program. Ten lines are generated between the ASET and ORG directives.

```
MILLENNIUM Z80 ASM V3.3                                                PAGE 1

00001                      STRING S1 (80)    ;DEFINE STRING VARIABLE S1
                                             ;WITH 80-CHARACTER
                                             ;MAXIMUM

00002      0003  L1        EQU     3         ;DEFINE CONSTANT SYMBOL
                                             ;L1 TO EQUAL 3

00003      0004  L2        ASET    4         ;DEFINE VARIABLE SYMBOL
                                             ;L2 TO EQUAL 4




00005      0100 >          ORG     100H      ;STARTS OBJECT CODE OF
                                             ;NEXT INSTRUCTION AT 100H

00006 0100 7E              LD    A,(HL)      ;LOADS THE CONTENTS OF
                                             ;MEMORY POINTED TO BY THE
                                             ;HL REGISTER PAIR INTO
                                             ;REG. A

00007                      END               ;END OF PROGRAM

MILLENNIUM Z80 ASM V3.3   SYMBOL TABLE LISTING                         PAGE 2

STRINGS AND MACROS

  S1 - - - - - 0050 S

SCALARS

  L2 - - - - - 0004 V

% (default SECTION (0101)

  L1 - - - - - 0

7 SOURCE LINES    7 ASSEMBLED LINES    1000 BYTES AVAILABLE
```

TITLE (Continued)

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	TITLE	{string expression}	[;charstring]

Purpose

The TITLE directive creates a text line at the top of each listing page for program identification.

Explanation

The character string specified as the TITLE operand is printed in the page heading between the assembler version number and the page number. As many as 31 characters may be entered. Any characters exceeding the 31-character limit are truncated. The actual TITLE directive is not printed on the listing.

Example

Assume the following TITLE statement is entered in a source program:

```
LABEL       OPERATION       OPERAND

            TITLE           "THIS IS THE PROGRAM TITLE"
```

Upon assembly, the specified TITLE appears within the heading at the top of each listing page of the program as follows:

```
MILLENNIUM Z80 ASM VX.X THIS IS THE PROGRAM TITLE                PAGE 1
```

STITLE (Continued)

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	STITLE	{string expression}	[;charstring]

Purpose

The STITLE directive creates a text line at the top of each listing page heading for program identification.

Explanation

The character string specified as the STITLE operand is printed between the page heading and the first source code line. A blank line is automatically inserted between the string and the beginning of the source code. As many as 72 characters may be entered. Any characters exceeding the 72-character limit are truncated. The actual STITLE directive is not printed on the listing.

Example

Assume the following TITLE statement is entered in a source program:

LABEL	OPERATION	OPERAND
	STITLE	"THIS LINE DEMONSTRATES STITLE USAGE"

Upon assembly, the specified STITLE line appears within the heading at the top of each listing page ~~of the program~~ as follows:

```
MILLENNIUM Z80 ASM VX.X                                      PAGE 1
(blank line)   THIS LINE DEMONSTRATES STITLE USAGE
.
. (source code)
.
.
```

WARNING

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	WARNING		[message]

Purpose

When an error is suspected within source code, the WARNING directive can be entered to generate an error message at assembly time. Thus, the nature of the errors in a program can be described upon assembly and listing.

Explanation

A warning message may be entered as a comment in the WARNING directive. Unlike other comments, the warning message is not preceded by a semicolon. Upon assembly, this optional message is printed on the assembly listing and on the output device, flagging the suspected error. The following assembler message is also displayed on both the assembler listing and the output device during assembly, below the specified warning message:

```
***** ERROR 0001:
```

Example

Assume the following WARNING directive is entered within a source program below a line containing an error.

LABEL	OPERATION	COMMENT
	WARNING	**** ENTRY OUT OF SEQUENCE

WARNING (Continued)

Upon assembly, the specified warning line appears below the source line containing the error. The message, ***** ERROR 001, also appears below the specified warning message.

```
.
.
.
000C 0003 - LEN ASET NCHR("ABB")
000D        WARNING                    ****ENTRY OUT OF SEQUENCE
*****ERROR: 001
.
.
.
```

Symbol Definition Directives

The assembler symbol definition directives are presented in the order shown in the following summary.

Mnemonic	Purpose
EQU	Permanently assigns a value to a symbolic name.
.STRING	Declares the named statement symbols as string variables.
ASET	Assigns or reassigns an expression's value to a string or numeric variable symbol.

EQU

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
symbol	EQU	{expression}	[;charstring]

Purpose

The EQU directive permanently assigns a value to a symbolic name.

Explanation

The symbol in the label field of an EQU directive is the symbolic name and the expression in the operand field represents the value. The symbol acquires the same base as the operand expression. No redefinition of this symbol is permitted.

The EQU directive operand field may contain a forward reference to a symbol label if the symbol does not appear in the operand field of an ORG, BLOCK, or another EQU directive.

If a symbol is declared in a GLOBAL directive and is defined by an EQU directive, the expression in the operand field of the EQU directive may not contain a HI, LO, or ENDOF function applied to an address. An error results when this occurs.

Example

The following line demonstrates EQU directive usage:

```
LABEL      OPERATION      OPERAND      COMMENT

L1         EQU            3            ;ASSIGNS THE VALUE 3 TO THE
                                       ;CONSTANT SYMBOL L1.
```

STRING

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	STRING	{{strvar1} [(lenexp1)]} [{,strvar2} [(lenexp2)]]...	[;charstring]

Purpose

The STRING directive declares the symbols named in the statement to be string variables.

Explanation

The STRING directive declares the symbols "strvar1" and "strvar2" to be string variables. A string variable is a symbol with an associated string value. Numeric expressions "lenexp1" and "lenexp2" may be optionally entered next to the string variables to specify the maximum character length of the values stored in the string variables. This maximum character length must be a scalar value greater than or equal to zero. When the optional character length expression is not specified, an eight-character maximum length is assumed. If the optional character length expression is specified, it must be enclosed within parentheses. An operand symbol named in a statement containing the optional character length expression must not be a forward reference.

A symbol must be declared with the STRING directive before it may be used as a string variable. Symbols declared as string variables must not be used for any other purpose within a program. Any number of string variables may be declared with the STRING directive. When a string variable is initially declared, its value is the same as that of the null string.

STRING (Continued)

Examples

The following examples demonstrate STRING directive usage:

LABEL	OPERATION	OPERAND	COMMENT
	STRING	STR(14)	;DECLARES STR ;AS A STRING ;VARIABLE WITH ;A MAXIMUM ;CHARACTER ;LENGTH OF 14
	STRING	A1,A2,A3,A4,X(NCHR("1234")	;DECLARES A1 ;THROUGH A4 ;AS STRING ;VARIABLES ;WITH A ;MAXIMUM ;CHARACTER ;LENGTH OF 8. ;DECLARES X AS ;A 4-CHARACTER ;STRING ;VARIABLE SINCE ;THE NUMBER ;OF CHARACTERS ;IN "1234" IS 4.

ASET

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
symbol	ASET	{expression}	[;charstring]

Purpose

The ASET directive is used to assign or reassign an expression value to a string or numeric variable symbol.

Explanation

The string or numeric variable symbol is entered in the label field of an ASET directive. A string variable symbol must have first been defined with the STRING directive. A numeric variable symbol must not have been previously defined, unless by another ASET directive. Variable symbols may not be subsequently redefined as labels, or be redefined by an EQU, STRING, SECTION, COMMON RESERVE, GLOBAL, OR MACRO directive. The value of a variable symbol may, however, be redefined by another ASET directive.

The expression value is entered in the operand field. The expression is then evaluated and the value is assigned to the variable symbol.

If an ASET directive contains a string-valued symbol and a numeric-valued expression, the numeric expression is converted to a string. This conversion is valid only when the numeric expression is a scalar value. The decimal value of the numeric expression is assigned to the string-valued symbol. The assigned string is six characters long, with the leftmost character being a minus sign if the value is negative. All numeric values are prefixed with leading zeros if less than six characters long. The numeric-expression to string-symbol conversion process is diagrammed as follows:

LABEL	OPERATION	OPERAND	COMMENT
string	ASET	numeric	;RESULTS IN EXPRESSION ;CONVERSION TO STRING

ASET (Continued)

If the ASET directive contains a numeric-valued symbol and a string-valued expression, the string expression is converted to a numeric value. Refer to Chapter 2 of this manual, ASSEMBLER SOURCE MODULE FORMAT, which describes String to Numeric Conversion. The string-expression to numeric-symbol conversion process is diagrammed as follows:

LABEL	OPERATION	OPERAND	COMMENT
numeric	ASET	string	;RESULTS IN EXPRESSION ;CONVERSION TO NUMERIC

Conversion is not required when a string-valued symbol is set to a string expression or a numeric-valued symbol is set to a numeric expression. When a symbol is set to an expression value, the symbol acquires the same section as the expression.

For string variable symbols where the length of the resulting expression value exceeds the maximum symbol string length, the expression value is truncated on the right before assignment. A truncation error code is then displayed.

Examples

Examples of typical ASET instructions and the resulting string-valued symbol expression values follow:

LABEL	OPERATION	OPERAND	COMMENT
	STRING	A1,A2(2),A3(45),A4(0)	;DEFINES STRING VARIABLE ;A1 WITH A DEFAULTING ;VALUE LIMIT OF 8 ;CHARACTERS. DEFINES ;STRING VARIABLES A2, A3, ;AND A4 WITH RESPECTIVE ;VALUE LIMITS OF 2, 45, AND ;0 CHARACTERS
A1	ASET	"AB"	;VALUE OF A1 IS "AB"
A2	ASET	A1	;VALUE OF A2 IS "AB"

(Program continued on next page)

ASET (Continued)

LABEL	OPERATION	OPERAND	COMMENT
A4	ASET	A1:A2	;VALUE OF A4 IS "", ;TRUNCATION ERROR SINCE ;A4 ALLOWS A VALUE OF ;ONLY 0 CHARACTERS
A3	ASET	"A MEDIUM LONG STRING"	;VALUE OF A3 IS "A MEDIUM ;LONG STRING" ;0 CHARACTERS
A1	ASET	A3	;VALUE OF A1 IS "A MEDIUM", ;TRUNCATION ERROR

The following example demonstrates string-to-numeric and numeric-to-string expression conversion.

LABEL	OPERATION	OPERAND	COMMENT
	STRING	A1,A2	;DEFINES STRING VARIABLES ;A1 AND A2
A1	ASET	14	;VALUE OF A1 IS "000014"
A2	ASET	-1	;VALUE OF A2 IS "-00001"
A1	ASET	5EH	;VALUE OF A1 IS "000094"
B1	ASET	2	;NUMERIC SYMBOL, B1, IS SET ;TO THE NUMERICALLY ;CONVERTED EXPRESSION, A2. ;TRUNCATION ERROR OCCURS, :SINCE A2 IS GREATER THAN ;TWO CHARACTERS (-00001). ;THE TWO RESULTING ;LEFTMOST ASCII CHARACTERS ;ARE -0, GIVING B1 A ;NUMERIC ASET VALUE OF ;2D30H

ORG

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	TITLE	{[/] expression}	[;charstring]

Purpose

The ORG directive sets the contents of the assembler location counter to either the address specified by the operand expression, the next address divisible by the operand expression, or the next odd address.

Explanation

Omission of the optional / (slash) operator sets the location counter to the address specified by the operand expresion. For example, when the following ORG directive is entered, the next instruction in the program begins at location 100H in the current section.

```
ORG 100H
```

If an ORG directive is omitted at the beginning of a program, the assembler location counter is set to 0. Usage of the / operator in the operand field causes the location counter to be set to the next location divisible by the operand expression. For example, when the current location counter contains 100H and the following ORG directive is entered, the next instruction begins at location 111H. (The next location divisible by 15H is 111H.)

```
ORG /15H
```

If the current location counter is divisible by the operand condition when the / operator is present, the location counter is unaffected. If the operand expression is "/0", the location counter is set to the next odd value. For example, then the current location counter contains 100H, and the following ORG directive is entered, the next instruction begins at location 101H.

```
ORG /0
```

ORG (Continued)

If the current location counter is already set to an odd value when the "/0" operand is entered, the location counter is unaffected.

The optional / operator may be used only with scalar-valued operand expressions.

Use care when entering the / operator, since the expected results may not be retained upon linking. For example, if ORG /0 is entered, the Linker puts the section containing this directive on an odd address, the ORG result is on an even address. This problem can be corrected by using the LOCATE command in the Linker. (Refer to the 9520 Software Development System Users Manual.)

Any symbol contained in the operand expression must have been defined in the label field of a previous statement in the program. If the operand expression contains a symbol previously defined in the label field of an EQU directive, the operand field of that EQU directive must not contain forward-referenced symbols.

A label symbol is generally not entered with this statement; however, if used, the symbol represents the resulting value of the location counter.

Example

The following ORG statement causes the object code generated by the next instruction to begin at location 100H.

LABEL	OPERATION	OPERAND	COMMENT
.			
.			
.			
	ORG	100H	;STARTS OBJECT CODE OF ;NEXT INSTRUCTION AT 100H
L1	LD	A,(HL)	;LOADS THE CONTENTS OF ;MEMORY POINTED TO BY THE ;HL REGISTER PAIR INTO ;REG. A
.			
.			
.			

ORG (Continued)

Upon assembly, the listing lines for the preceding instructions appear as follows:

```
.
.
.
00005           0100 >         ORG 100H           ;STARTS OBJECT CODE OF
                                                  ;NEXT INSTRUCTION AT 100H

00006   0100    7E        L1   LD  A,(HL)         ;LOADS THE CONTENTS OF
                                                  ;MEMORY POINTED TO BY THE
                                                  ;HL REGISTER PAIR INTO
                                                  ;REG. A
.
.
.
```

Notice the relocation indicator (>) on line 00005. The LD instruction object code begins at location 100H.

Data Storage Control Directives

The assembler data storage control directives appear in the order shown in the following summary.

Mnemonic	Purpose
BYTE	Allocates one byte of memory to each expression specified in the operand field.
WORD	Allocates two bytes of memory to each expression specified in the operand field.
ASCII	Stores ASCII text in memory.
BLOCK	Reserves a specified number of bytes in memory.

BYTE

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	BYTE	{expression} [,expression]	[;charstring]

Purpose

This directive allocates one byte of memory to each expression specified in the operand field.

Explanation

Each data byte is represented by an expression. The data is stored in the object module in the order in which it appears in the operand field. If more than one expression is specified in the operand field, the expressions are stored in consecutive bytes. The optional label field symbol represents the address of the first byte of data specified by the directive.

If the expression represents a value exceeding the eight-bit capacity, the eight least significant bits are used and a truncation error code is displayed. For example, a statement containing the following BYTE directive generates 32H upon assembly and issues a truncation error response.

LABEL	OPERATION	OPERAND	COMMENT
	BYTE	"K2"	;GENERATES 32H, ;TRUNCATION ERROR

Example

In the following BYTE directive, one byte of memory is allocated to the expression values 24 hexadecimal and 22 decimal. The label symbol, FSTBYT, represents the address of the first byte specified, 24H.

LABEL	OPERATION	OPERAND	COMMENT
FSTBYT	BYTE	24H,22	;ALLOCATES ONE BYTE OF ;MEMORY TO THE ;EXPRESSION VALUES 24H ;AND 22 DECIMAL

WORD

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	~~BYTE~~ WORD	{expression} [,expression]	[;charstring]

Purpose

~~This~~ The WORD directive allocates two bytes of memory to each expression specified in the operand field.

Explanation

This directive is identical to the BYTE directive except that two bytes of memory are allocated in the object module for every expression specified in the operand field. These two-byte values are stored in memory with the low byte first, followed by the high byte. If an expression represents a single byte value, the high byte is stored as zero. If more than one expression is specified in the operand field, the expressions are stored in consecutive words. The optional label field symbol represents the address of the first byte of data stored in memory.

WORD (Continued)

Example

In the following WORD directive, two bytes of memory are allocated to the expression values 356 and 427 decimal. The label symbol LABSYM represents the address of the first byte of the value 356 decimal.

LABEL	OPERATION	OPERAND	COMMENT
LABSYM	WORD	356,427	;ALLOCATES TWO BYTES OF ;MEMORY EACH TO THE ;EXPRESSION VALUES 356 AND ;427 DECIMAL

ASCII

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	ASCII	{string expression}[,string expression	[;charstring]

Purpose

The ASCII directive allows the user to store text in memory easily.

Explanation

ASCII characters may be specified in the operand field in the form of a string expression. If more than one operand is specified on a line, each operand is separated by a comma. The optional label symbol represents the memory address allocated to the first operand field character.

ASCII (Continued)

Examples

Assume the following lines of source code reside on disk:

```
LABEL      OPERATION    OPERAND                          COMMENT
.
.
.
           ASCII        "HELLO","GOODBYE"                ;PUTS HELLO AND
                                                         ;GOODBYE IN OBJECT
                                                         ;MODULE AS ASCII
                                                         ;CODE

           ASCII        "BYE"                            ;PUTS BYE IN OBJECT
                                                         ;MODULE AS ASCII
                                                         ;CODE

           ASCII        ""                               ;PUTS NULL STRING
                                                         ;IN OBJECT MODULE
                                                         ;AS ASCII CODE

           STRING       STR1 (20)                        ;DEFINES STR1 AS
                                                         ;STRING VARIABLE
                                                         ;WITH A MAXIMUM
                                                         ;CHARACTER LIMIT
                                                         ;OF 20

STR1       ASET         "ABCDEF"                         ;ASSIGNS ASCII
                                                         ;VALUE OF ABCDEF
                                                         ;TO STR1

           ASCII        STR1                             ;PUTS ABCDEF IN
                                                         ;OBJECT MODULE AS
                                                         ;ASCII CODE

           ASCII        STR1:" ":STRING(NCHR(STR1))      ;PUTS ABCDEF, A
                                                         ;BLANK, AND THE
                                                         ;NUMBER OF
                                                         ;CHARACTERS IN
                                                         ;ABCDEF (6) IN
                                                         ;OBJECT MODULE AS
                                                         ;CONCATENATED
                                                         ;ASCII CODE
.
.
.
```

ASCII (Continued)

The hexadecimal object code generated by the string expressions in the preceding source code is shown as follows:

SOURCE	OBJECT
"HELLO","GOODBYE"	48454C4C4F474F4F44425945
"BYE"	425945
""	(nothing)
"ABCDEF" (string value of STR1)	414243444546
"ABCDEF 000006"	41424344454620303030303036

For hexadecimal and ASCII conversion tables, refer to Appendix C.

BLOCK

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	BLOCK	{expression}	[;charstring]

Purpose

The BLOCK directive reserves a specified number of bytes in memory.

Explanation

The BLOCK operand expression indicates the number of bytes to reserve in memory. The operand expression must be a positive value. The operand expression must be either a numeric or string constant, or a symbol. If the operand expression contains a symbol, the symbol must be previously defined in the program. Additionally, if the symbol is defined by the EQU directive, that EQU directive's operand field must conform to these same rules. The expression specified in the BLOCK operand must be a scalar value.

Example

The following BLOCK directive reserves a 32-byte memory storage block:

LABEL	OPERATION	OPERAND	COMMENT
	BLOCK	~~32+~~ 32	;RESERVES 32 BYTES OF :MEMORY

BLOCK (Continued)

Macro Definition Directives

The macro definition directives are presented in the order shown in the following summary. A complete description of macro capability is presented in Chapter 5.

Mnemonic	Purpose
MACRO	Defines the name of a source code block used repeatedly within a program.
ENDM	Terminates the macro definition block.
REPEAT	Enables the macro lines following the REPEAT statement up to the ENDR statement to be assembled repeatedly.
ENDR	Signals the corresponding REPEAT block termination.
INCLUDE	Inserts text from a specified file into the program.

MACRO

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	MACRO	{symbol}	[;charstring]

Purpose

The MACRO directive defines the name of a source code block used repeatedly within a program.

Explanation

A macro is a shorthand method for inserting a block of source code into a program one or more times. The MACRO directive names the source code block to be inserted into the main program. The symbolic macro name appears in the operand field of the MACRO directive, and is later used as a reference when the source code block is called for insertion during assembly. The block of source code to be inserted is called the macro definition block, and immediately follows the MACRO directive. The macro definition block terminates with an ENDM directive. When the macro name appears within the operation field of the main program during assembly, the macro definition block is inserted and assembled within the main program. This process is called macro expansion.

The symbolic macro name and the macro definition block are generally defined at the beginning of a user program. The macro name and definition block must be defined prior to the initial macro definition block usage.

For a further description of macro capability and usage, refer to Chapter 5.

MACRO (Continued)

Example

The MACRO directive below defines the block of macro code following the directive.

LABEL	OPERATION	OPERAND	COMMENT
	MACRO	MACRNAME	;DEFINES MACRNAME AS MACRO ;NAME
	BYTE	3,5,1	;ALLOCATES ONE BYTE OF ;MEMORY EACH TO THE CONSTANT ;VALUES 3, 5, AND 1
	WORD	2	;ALLOCATES TWO BYTES OF ;MEMORY TO THE CONSTANT ;VALUE 2
	ENDM		;END OF MACRO DEFINITION ;MACRNAME

•
•
•

Later statements in this program may call the macro definition block whenever the specified BYTE and WORD statement sequence is desired.

ENDM

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	ENDM		[;charstring]

Purpose

The ENDM directive signals the end of a macro definition block.

Explanation

When an ENDM directive is encountered in a macro definition block, the macro is terminated and assembly continues with the next statement in the program following the macro call.

Example

The following ENDM directive terminates the macro definition block named NUMNAK.

LABEL	OPERATION	OPERAND	COMMENT
	MACRO	NUMNAK	;DEFINES NUMNAK AS MACRO ;NAME
	BYTE	3,27,22	;ALLOCATES ONE BYTE OF ;MEMORY TO THE CONSTANT ;VALUES 3, 27, AND 22
	WORD	255	;ALLOCATES TWO BYTES OF ;MEMORY TO THE CONSTANT ;VALUE 255 ;END OF MACRO DEFINITION

REPEAT - ENDR

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	REPEAT	{expression1} [,expression2]	[;charstring]
[symbol]	ENDR		[;charstring]

Purpose

The REPEAT directive enables the macro lines following the REPEAT directive, up to the ENDR directive, to be assembled repeatedly. The ENDR directive signals the end of each repeat cycle.

Explanation

When a REPEAT directive is encountered upon macro expansion, the first expression specified in the operand field is evaluated. The lines up to the ENDR directive are ignored when the REPEAT operand, "expression1" is equal to zero (false). If the expression is true (non-zero), the lines up to the ENDR directive are assembled repeatedly until the expression does equal zero, or the maximum number of repeat cycles is exceeded. The second operand "expression2" may be optionally entered to specify the maximum number of repeat cycles. If the maximum number of repeat cycles is not specified, the value of "expression2" defaults to 255. Attempts to repeat beyond the value of "expression2" causes an error code to be displayed. Both operand expressions must be scalar values.

REPEAT - ENDR blocks may be nested. The nesting depth is limited only by the amount of memory available to the assembler. Each REPEAT condition must be properly nested, thus having a matching ENDR occurring within the scope of that particular REPEAT condition. REPEAT - ENDR blocks may not cross the boundary of a macro expansion or of an IF - ENDIF block. A REPEAT - ENDR block is valid only within a macro definition block.

REPEAT - ENDR (Continued)

Example

The example that follows demonstrates REPEAT - ENDR block usage within a macro named CONDRID.

LABEL	OPERATION	OPERAND	COMMENT
	MACRO	CONDRID	;DEFINES CONDRID AS MACRO ;NAME
AGAIN	ASET	1	;INITIALIZES AGAIN TO EQUAL ;1 AT ASSEMBLY TIME
	REPEAT	AGAIN < = 27	;REPEAT WHILE AGAIN IS LESS ;THAN OR EQUAL TO 27
	BYTE	AGAIN	;GENERATES ONE BYTE OF ;MEMORY TO AGAIN
AGAIN	ASET	AGAIN + 1	;INCREMENT AGAIN AT ;ASSEMBLY TIME
	ENDR		;END OF REPEAT CONDITION
	BYTE	0DH	;GENERATES CARRIAGE ;RETURN
	ENDM		;END OF MACRO DEFINITION

INCLUDE

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	INCLUDE	{string expression}	[;charstring]

Purpose

The INCLUDE directive is used to insert text from a specified source file into a program.

Explanation

When the INCLUDE directive is encountered, text from the file specified in the operand field is inserted into the program. If the INCLUDE directive is contained in a macro body, the text file is inserted at macro expansion time. Parameters within the included file cannot reference arguments used in the containing macro. Refer to Chapter 5 for a discussion of text substitution within macros. The text file specified by the INCLUDE directive may not terminate a MACRO, REPEAT or IF block. Additionally, the text may not contain another INCLUDE directive.

An INCLUDE directive may also be used within normal source code, outside of macro definition blocks. When this occurs, the inserted text may contain macro definitions.

INCLUDE (Continued)

Example

The following example demonstrates INCLUDE directive usage.

```
LABEL    OPERATION    OPERAND          COMMENT
         .
         .
         .
         INCLUDE      "B:FILEA.SRC"    ;INSERTS FILE A OF DRIVE B INTO
                                       ;THE CURRENT PROGRAM AT THE
                                       ;ADDRESS OF THE CURRENT
                                       ;LOCATION COUNTER.

         INCLUDE      "A:FILEB.SRC"    ;INSERTS FILE B OF DRIVE A INTO
                                       ;THE CURRENT PROGRAM AT THE
                                       ;ADDRESS OF THE CURRENT
                                       ;LOCATION COUNTER.

         INCLUDE      "FILEC.SRC       ;INSERTS FILE C OF DRIVE A INTO
                                       ;THE CURRENT PROGRAM AT THE
                                       ;ADDRESS OF THE CURRENT
                                       ;LOCATION COUNTER.
```

NOTE: The third INCLUDE statement specifies FILEC.SRC. The default logical drive designator "A" will be prefixed, making it A:FILEC.SRC.

Conditional Assembly Directives

The conditional assembly directives are presented in the order shown in the following summary.

Mnemonic	Purpose
IF	Causes the assembly of the source code lines following the IF directive, up to the ENDIF directive, when the specified operand expression is true (non-zero).
ELSE	Causes an alternate source block to be assembled when the containing IF expression is false.
ENDIF	Signals the corresponding IF block termination.
EXITM	Terminates the current macro expansion before encountering an ENDM directive.

IF - ELSE - ENDIF

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	IF	{expression}	[;charstring]
[symbol]	ELSE		[;charstring]
[symbol]	ENDIF		[;charstring]

Purpose

The IF directive causes assembly of the source code lines following the IF directive, up to the ENDIF (or ELSE, if present) directive, when the specified operand expression is true. The ELSE directive causes an alternate source block to be assembled when the containing IF expression is false. ENDIF signals the corresponding IF block termination.

Explanation

When an IF directive is encountered, the expression specified in operand field is evaluated. If the result of the expression is zero (false) source lines between the IF and ENDIF directives are ignored (not assembled). The ENDIF directive then terminates the condition. If the result of the expression is non-zero (true), the source lines are assembled once normally.

An optional ELSE directive block may be nested within the IF source block. If an ELSE block is present, a false IF expression causes assembly of the source lines from the ELSE directive up to the ENDIF directive. The ELSE block is ignored when the expression in the IF directive operand field is true. Only one ELSE directive is allowed within each IF-ENDIF block.

IF - (ELSE) - ENDIF blocks may be nested as deeply as desired, limited only by the amount of memory available to the assembler. Each IF directive must be properly nested thus having a matching ENDIF occurring within the scope of that particular IF condition. IF - (ELSE) - ENDIF blocks may not cross the boundaries of REPEAT - ENDR blocks, macro expansions, and other IF - (ELSE) - ENDIF blocks.

IF - ELSE - ENDIF (Continued)

Examples

The following example demonstrates IF - (ELSE) - ENDIF block usage:

```
LABEL     OPERATION     OPERAND          COMMENT

          IF            " '1' " = ""     ;CHECKS TO SEE IF THE FIRST
                                         ;MACRO ARGUMENT IS
                                         ;UNDEFINED
          WORD          0F7H             ;IF SO, GENERATES A WORD
                                         ;CONTAINING 0F7H
          ELSE                           ;OTHERWISE
          WORD          '1'              ;GENERATES A WORD
                                         ;CONTAINING THE FIRST
                                         ;ARGUMENT
          ENDIF                          ;END OF IF CONDITION
```

The following example demonstrates nested IF - (ELSE) - ENDIF block usage:

```
LABEL     OPERATION     OPERAND          COMMENT

          IF            " '1' "<>""      ;CHECKS TO SEE IF THE FIRST
                                         ;MACRO ARGUMENT
                                         ;EXISTS
          IF            '1'< 0F0H        ;IF SO, CHECKS TO SEE IF THE
                                         ;FIRST MACRO ARGUMENT IS
                                         ;LESS THAN 0F0H
          WORD          0F7H - '1'       ;IF SO, GENERATES ONE WORD
                                         ;CONTAINING THE DIFFERENCE
                                         ;BETWEEN 0F7H AND THE
                                         ;FIRST ARGUMENT
          ELSE                           ;OTHERWISE, IF FIRST
                                         ;ARGUMENT IS GREATER
                                         ;THAN 0F0H...
          WORD          '1'              ;GENERATES ONE WORD
                                         ;CONTAINING FIRST MACRO
                                         ;ARGUMENT
          ENDIF                          ;END OF INNER IF CONDITION
          ELSE                           ;OTHERWISE, IF THE
                                         ;ARGUMENT DOES NOT EXIST...
          WORD          0F7H             ;GENERATE A WORD
                                         ;CONTAINING 0F7H
          ENDIF                          ;END OF OUTER IF CONDITION
```

EXITM

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	EXITM		[;charstring]

Purpose

The EXITM directive terminates the current macro expansion before encountering an ENDM directive.

Explanation

EXITM is generally used within IF - (ELSE) -ENDIF and REPEAT - ENDR blocks to conditionally terminate macro expansions. EXITM is valid only within a macro definition block.

Example

The following ENDM directive terminates the macro definition block named NUMNAK.

LABEL	OPERATION	OPERAND	COMMENT
	MACRO	CONDMAC	;DEFINES CONDMAC AS MACRO ;NAME
	BYTE	1,2,0	;ALLOCATES ONE BYTE OF ;MEMORY FOR EACH OF THE ;THREE VALUES 1, 2, AND 0
	IF	" '3' "=""	;TESTS TO DETERMINE IF ;3RD PARAMETER IN ;MACRO CALL EXISTS
	BYTE	255	;IF 3RD ARGUMENT DOES NOT ;EXIST, ONE BYTE IS ALLOCATED ;CONTAINING 255 DECIMAL
	EXITM		;TERMINATES MACRO ;EXPANSION IF CONDITION IS ;SATISIFED
	ENDIF		;END OF IF CONDITION
	BYTE	'3'	;OTHERWISE, ONE BYTE IS ;ASSIGNED CONTAINING THIRD ;ARGUMENT
	ENDM		;END OF MACRO DEFINITION

Section Definition Directives

The section definition directives appear in this subsection in the order shown in the summary below. Relocation options used with the section definition directives follow this summary. For a discussion of the methods by which the Linker relocates sections, refer to the 9520 Software Development System Users Manual.

Mnemonic	Purpose
SECTION	Declares a Linker section, assigns a section name, and defines the section parameters.
COMMON	Declares a Linker section, assigns a section name, and defines the section type to be common.
RESERVE	Sets aside a work space in memory. Upon linking, all reserve sections with the same name are concatenated into a single reserve section.
RESUME	Continues the definition of code for a given section.
GLOBAL	Declares one or more symbols to be global variables.
NAME	Declares the name of an object module.

RELOCATION OPTIONS

The PAGE, INPAGE, or ABSOLUTE option may be specified in the operand field, to direct the relocation of a block of code in the SECTION and COMMON directives. The PAGE or INPAGE option is also available to the RESERVE directive. When options are not specified, the section is relocated on any byte address. The effects of these options are summarized as follows:

PAGE - Causes the section to be relocated at the starting address of a physical block of memory. This block of memory, also called a "page", is 256 bytes long with a starting address that is evenly divisible by this length. Therefore, the starting address of a page may be 0,256,512, etc.

INPAGE - Causes the section to be relocated on any byte address provided the section does not extend across page boundaries.

ABSOLUTE - Causes the memory allocation to be the actual areas specified by the ORG directives at assembly time. (No relocation of this section is performed.) Arithmetic functions performed on addresses defined in absolute sections are subject to the same restrictions as addresses performed on relocatable sections. Refer to Section 2 describing Binary Arithmetic Operators.

If no option is entered with the section definition directives, the specified section is byte relocatable, indicating a lack of restrictions on where the Linker may place the section.

SECTION

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	SECTION	{symbol} [,PAGE / ,INPAGE / ,ABSOLUTE]	[;charstring]

Purpose

The SECTION directive is used to declare a program section, assign the section a name, and define its parameters.

Explanation

All program text following the SECTION directive, up to the next SECTION, COMMON, or RESUME directive, is defined to be a program section. All text within a program section is assembled with the same location counter, and hence, has the same base. Each section has a separate location counter and must be relocated as a block. The initial value of the location counter for a given section is 0. The symbol specified in the SECTION operand field is the section name, and is a global symbol. The section name must be unique to each assembly and, therefore, cannot appear in multiple SECTION directives. When separate object modules containing sections with the same name are linked, an error is generated.

The optional second operand in the SECTION directive can be used to place restrictions on the relocatability of the section. (Refer to previous discussion on Relocation Options in this subsection.) If no option is specified, the Linker considers the section to be byte relocatable.

SECTION (Continued)

When a label symbol is entered on the SECTION directive, the symbol represents address 0, the initial value of the resulting section's location counter. Additionally, the declared section name in the operand field may be used as a normal global symbol, and referenced in the operand field of other statements throughout the assembly. The section name has the same value as the label on the SECTION directive.

Example

The following source line demonstrates SECTION directive usage.

```
LABEL      OPERATION       OPERAND           COMMENT

           .
           .
           .
           SECTION         SEC1              ;GENERATES BYTE
           .                                 ;RELOCATABLE SECTION,
           .                                 ;SEC1
```

COMMON

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	COMMON	{symbol} [,PAGE / ,INPAGE / ,ABSOLUTE]	[;charstring]

Purpose

The COMMON directive declares a section, associates a name with the section, assigns the section parameters, and defines the section type to be common.

Explanation

The COMMON directive performs the same functions as the SECTION directive, except that the same name may identify common sections in more than one source module. Common sections with the same name are relocated at the same address by the Linker. Each section with the same name should specify the same relocation option; otherwise, the desired relocation might not result at link time. The Linker allocates enough memory to contain the largest of the common sections with the same name.

This section type is modeled after the COMMON area of FORTRAN.

Example

The following example demonstrates COMMON directive usage.

```
LABEL     OPERATION      OPERAND        COMMENT

          .
          .
          .
          COMMON         WRKAREA        ;DEFINES WRKAREA AS A COMMON
                                        ;SECTION.  IF WRKAREA EXISTS
                                        ;IN MULTIPLE OBJECT MODULES.
                                        ;LINKER CHOOSES THE LARGEST
                                        ;SECTION NAMED WRKAREA FOR
                                        ;MEMORY ALLOCATION
```

RESERVE

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	RESERVE	{symbol, expression} [,PAGE ,INPAGE]	[;charstring]

Purpose

The RESERVE directive is used to set aside a workspace in memory. Upon linking, all reserved workspaces (sections) with the same name are combined into a single section.

Explanation

The symbol in the operand field of the RESERVE directive is the assigned name of the section. The operand expression specifies the number of bytes to be reserved for the current object module. The expression must be a scalar value. The RESERVE directive does not change the current section.

More than one object module may contain reserve sections of the same name. The length of the reserve section allocated by the Linker is the sum of all reserve sections with the same name.

RESERVE (Continued)

Example

The following example demonstrates section space allocation with the RESERVE directive.

```
LABEL   OPERATION   OPERAND            COMMENT
        .
        .
        .
        RESERVE     BNCHCODE,100H      ;RESERVES A SECTION DEFINED
                                       ;AS BNCHCODE AND
                                       ;ALLOCATES 256 BYTES OF
                                       ;MEMORY TO BE ADDED TO THE
                                       ;SIZE OF BNCHCODE
        .
        .
        .
        WORD        BNCHCODE           ;PLACES ONE WORD IN THE
                                       ;CURRENT SECTION HAVING
                                       ;THE ADDRESS OF THE
                                       ;BEGINNING OF THE BNCHCODE
                                       ;SECTION
        WORD        ENDOF(BNCHCODE)    ;PLACES ONE WORD IN THE
                                       ;CURRENT SECTION HAVING
                                       ;THE ENDING ADDRESS OF
                                       ;BNCHCODE
```

RESUME

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	RESUME	[symbol]	[;charstring]

Purpose

The RESUME directive continues the definition of a given section.

Explanation

The RESUME directive continues the definition of the section specified by the optional operand symbol. If no operand symbol is used, the definition of the default section is continued. Any source code that is not preceded by a SECTION or COMMON directive is included in the default section. The name given to the default section is a percent sign (%) followed by the object file name. When no object file is present, the name given to the default section is %.

If used, the label symbol is assigned the value of resumed section's location counter.

Example

The example that follows demonstrates section definition resumption with the RESUME directive.

```
LABEL     OPERATION    OPERAND        COMMENT
          .
          .
          SECTION      A31            ;DEFINES SECTION A31
          .
          .
          .
          SECTION      B31            ;DEFINES SECTION B31
          .
          .
          .
          RESUME       A31            ;RESUMES SECTION A31
          .
          .
          .
```

GLOBAL

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	GLOBAL	{symbol} [,symbol]...	[;charstring]

Purpose

The GLOBAL directive declares one or more symbols to be global variables. A global variable located in one source module may be referenced by another source module.

Explanation

Symbols specified in the GLOBAL directive operand field are designated to be global variables. Global variables defined in the current assembly are called bound globals. If the global variables are not defined in the current assembly, they are called unbound globals and their references must be resolved by the Linker.

The value of a global symbol must be unique within an assembly. A maximum of 254 names may be defined to be global variables. This maximum includes all names used in SECTION, COMMON, RESERVE, and GLOBAL directives.

GLOBAL (Continued)

Example

The following example demonstrates definition of global variables with the GLOBAL directive.

LABEL	OPERATION	OPERAND	COMMENT
	•		
	•		
	•		
	GLOBAL	HIGUY,BYEGUY	;DEFINES THE SYMBOLS HIGUY ;AND BYEGUY TO BE USED AS ;GLOBAL SYMBOLS
	•		
	•		
	•		
HIGUY	EQU	$	;HIGUY IS EQUIVALENT TO ;CURRENT LOCATION :COUNTER
	CALL	BYEGUY	;JUMPS TO SUBROUTINE ;BYEGUY DEFINED IN ;ANOTHER ASSEMBLY
	•		
	•		
	•		

NAME

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	NAME	{symbol}	[;charstring]

Purpose

The ~~INCLUDE~~ NAME directive declares the name of an object module ~~is used to insert text from a specified source file into a program.~~

Explanation

The symbol in the operand field of the NAME directive is the name assigned to the object module. If more than one NAME directive appears within an assembly, only the first NAME directive is used; the rest are ignored.

Note that the object module name, as declared by the NAME directive, is distinct from the file name that the object module is stored under. Note also that the default section derives its name from the object file, not the NAME directive.

Example

The following example demonstrates the object module naming with the NAME directive.

```
LABEL     OPERATION     OPERAND          COMMENT

          .
          .
          .
          NAME          ~~"~~XMPLSUB~~"~~        ;NAMES OBJECT MODULE
                                         ;XMPLSUB
          .
          .
          .
```

END

MODULE TERMINATION DIRECTIVE

SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	END	[expression]	[;charstring]

Purpose

The END directive terminates source modules.

Explanation

The END directive terminates a source module contained in one or more disk files. A source module is also terminated when the end of the last input file is read. END directive usage is, therefore, optional.

The optional expression in the operand field represents the starting address for program execution, which is called a transfer address. If present, the specified operand value is placed in the object module and may be used by the MP/M or CP/M LOAD command when loading the object module into program memory. At link time, if more than one module has a transfer address, the first one encountered is used.

INTRODUCTION

A macro is a shorthand approach for inserting source code into a program. A macro is often used when the same, or nearly the same, code is repeatedly used within a program. A block of macro code is called a macro definition block. The source code that results from this block may be altered each time the macro is called so that the object code generated depends on the information specified in the macro call. The code generated by a macro call is called a macro expansion, since it results from, and is usually larger than, the macro called.

This section describes all phases of macro definition, calling, and expansion. The structure of this section closely follows the process leading up to macro expansion. First, an examination of the general macro expansion process is illustrated to provide a basis of understanding. An examination of each phase of the process is then presented in greater detail.

Basic Macro Expansion Process

The macro expansion process is illustrated in figure 5-1. A written explanation of the process follows the figure.

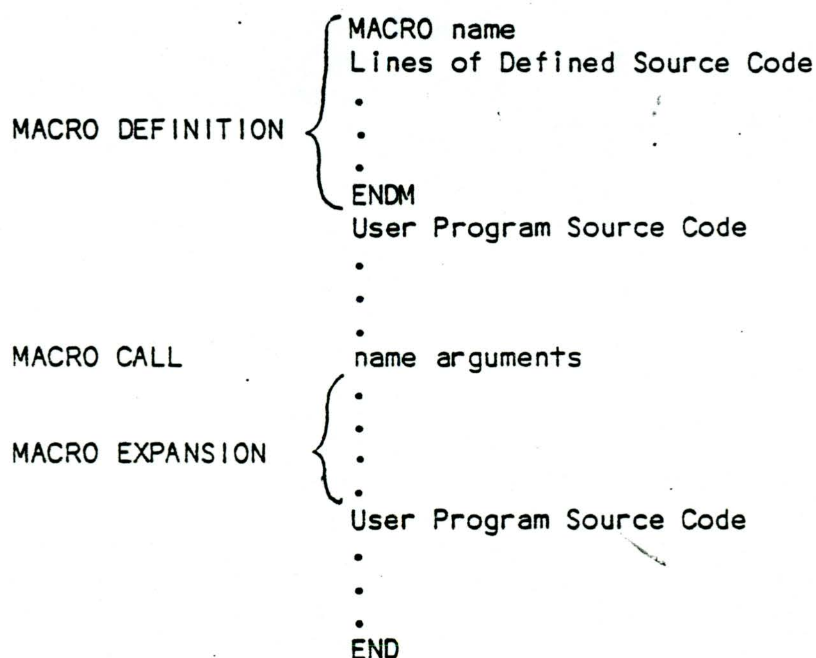

Figure 5-1. The Macro Expansion Process

As mentioned, there are three phases of macro usage: definition, calling, and expansion. First the macro must be defined. The macro is given a name followed by a body. The macro is defined in a macro definition directive. The macro body is called a macro definition block. The macro definition block is made up of source lines that are stored in unassembled form, until the macro is used. To use the macro, the programmer codes a macro call within a program. The macro name appears in the macro call directive's operation field. When the macro call is encountered during assembly, the macro definition block is inserted and assembled within the main program. This process is called macro expansion.

The user may alter any parameters used within the macro definition block by inserting corresponding arguments within the operand field of a macro call. One line at a time, the assembler replaces the specified parameters with corresponding arguments in the macro call. The assembler inserts the line from the macro definition block into the user program. The line is then assembled. This procedure repeats for each line in the macro definition block.

Macro Definition Directive

A macro is defined by first entering the macro definition directive in the following format. In this macro definition directive, "name" is the macro name that is later used as a reference for the macro call.

MACRO name

Macro Definition Directive Conventions

A macro is generally defined at the beginning of a program. A macro must always be defined prior to its initial use. A macro may not be defined within another macro definition block. A macro name is a symbol containing up to eight characters, the first character being alphabetic. The macro name must be unique from all symbols in a user program.

Macro Definition Block

The lines following the macro definition directive, up to and including an ENDM directive, become a pre-defined block of code referred to as a macro definition block. A macro definition block may contain any instruction or assembler directive (except the END and MACRO directives). A macro definition block may contain calls to other macros or even calls to itself. When a macro call occurs within another macro definition block, any replacement that may occur on the macro call is performed before the inner macro is called. A macro definition block may not contain the definition of another macro.

Source Code Alteration

An additional macro capability allows code to be altered within a macro definition block. Upon expansion, parameters within single quotes, serving as place holders in the macro definition block, are replaced by the arguments defined in a macro call.

In summation:

Parameters	-	are place holders within a macro-definition block.
Arguments	-	are values, defined within a macro call directive, that replace parameters.

Any numeric parameter surrounded by single quotes ('N') is replaced by the Nth argument passed to the current macro expansion. In the following BYTE directive, for example, the first argument passed to the current macro expansion is substituted for the first parameter, labeled '1', upon macro expansion.

```
BYTE 3,5,'1'
```

N may be either a number or a numeric-valued ASET symbol. An ASET symbol is assigned a value by the ASET directive. This capability is discussed in Chapter 4, ASSEMBLER DIRECTIVES, describing the ASET directive. If N is greater than the number of arguments provided, the null string is substituted. Text substitution may occur anywhere on a line.

Additional Special Macro Definition Characters

The following special characters are only available for use within macro definition blocks.

The @ Character

The "at" character, when surrounded by single quotes ('@'), provides unique labels for each macro expansion. The @ character is replaced by a four-character hexadecimal value that is unique within each macro call. In the example that follows, each time the macro is called, a unique four-character hexadecimal value replaces the @ character. The following statement creates a unique seven-character label.

LABEL	OPERATION	OPERAND
LAB'@'	EQU	$

The '@" Character (Continued)

The '@' in the preceding label is replaced by a number unique to the current macro call. This replacement prevents LAB from being defined more than once by subsequent macro calls.

The # Character (Continued)

The "pound" character, when surrounded by single quotes ('#') is replaced by a five-digit decimal number. The number represents the total number of arguments that are passed to the current macro expansion. In the example that follows, expansion of all lines of code within a REPEAT block continues until the total number of arguments passed is exceeded. Suppose three arguments are passed during expansion of the macro containing this code:

```
LABEL       OPERATION       OPERAND         COMMENT

            .
            .
            .
J           ASET            1               ;INITIALIZES J TO EQUAL 1
                                            ;AT ASSEMBLY TIME
            REPEAT          J <= '#'        ;REPEAT WHILE J IS LESS THAN
                                            ;OR EQUAL TO 3
J           ASET            J + 1           ;INCREMENT J
            .                               .
            .                               .
            ENDR                            ;END OF REPEAT CONDITION
            .
            .
            .
```

The % Character

The "percent" character, when surrounded by single quotes ('%'), is replaced by the name of the current section or common. The name is returned as a string. If the current section is the default section, the null string is returned.

The % Character (Continued)

In the example that follows, the percent sign character is used to represent the name of the current section.

LABEL	OPERATION	OPERAND	COMMENT
	STRING	SECNAM(8)	;DEFINES STRING, SECNAM, ;WITH EIGHT-CHARACTER ;MAXIMUM
SECNAM	ASET	" '%' "	;SECNAM IS SET TO NAME OF ;CURRENT SECTION
	SECTION	BBB	;DEFINES NEW SECTION BBB
	. . .		
	RESUME	'SECNAM'	;RESUMES PREVIOUS SECTION
	. . .		

The ↑ or ^ Character

The up-arrow (↑) or caret (^) character may be entered just prior to any character having special meaning, thus allowing the special character to be interpreted as a regular part of the text. The up-arrow (↑) or caret (^) character is available in all phases of the MILLENNIUM SYSTEMS Assembler and is described in the manner in which it affects macro definition. In the example that follows, the caret (^) character removes the special meaning of the single quote character.

LABEL	OPERATION	OPERAND
	ASCII	"THAT^'S ALL FOLKS."

Upon macro expansion, the following code is generated in memory.

THAT'S ALL FOLKS.

Macro Termination

A macro definition block is terminated by an ENDM statement.

Macro Calling

A macro is invoked when a macro call is encountered within a program. A macro call contains the macro name to be called in the statement's operation field as follows:

LABEL	OPERATION	OPERAND
	name	

Include Directive Text Insertion

Another method for calling text into a program involves INCLUDE directive usage. The INCLUDE directive (see Chapter 4, describing ASSEMBLER DIRECTIVES) may be used to insert text into a program from a specified file. The INCLUDE directive may be a part of a MACRO, IF - ENDIF, or REPEAT - ENDR block, as long as it does not terminate any of those blocks. The name of the file to be inserted is entered in the operand field of the INCLUDE directive as follows:

LABEL	OPERATION	OPERAND
	INCLUDE	string expression

Text Substitution

Optional arguments separated by commas within the operand field of the macro call define the values to replace the parameters within the block as the macro is expanded. For example, the following macro call invokes the macro named EVALC and defines the arguments 25 and ARG2 for substitution within the block of code as the macro is expanded.

LABEL	OPERATION	OPERAND	COMMENT
	EVALC	25,ARG2	;INVOKES MACRO EVALC AND ;DEFINES FIRST TWO ;ARGUMENTS FOR ;SUBSTITUTION WITHIN MACRO ;DEFINITION BLOCK AS 25 ;AND ARG2

The preceding example contains the following arguments:

Argument 1 = 25
Argument 2 = ARG2

A label appearing in a macro call is assigned the value of the location counter prior to macro expansion.

Special Macro Calling Characters

The following special function is available for use within macro calls.

The [] Construct

Square brackets [] may be used to group code for inclusion as an argument within a macro call. All characters enclosed within square brackets are considered to represent a single argument. Square brackets may not be nested. Unlike the argument resulting when a character string is enclosed within double quotes, the square brackets are not passed to the source text during macro expansion. For example, the following macro call parameters produce the corresponding arguments:

LABEL	OPERATION	OPERAND	COMMENT
	PNPDG	ABC,1,"ABC,1",[ABC,1]	;INVOKES MACRO ;PNPDG AND ;SUBSTITUTES THE ;ARGUMENTS ABC, ;1,"ABC,'",ABC,1

The preceding example contains the following arguments:

Argument 1 = ABC
Argument 2 = 1
Argument 3 = "ABC,1"
Argument 4 = ABC,1

The ↑ or ^ Character

The up-arrow (↑) or caret (^) character may be entered just prior to any character having special meaning, thus allowing that character to be interpreted as a regular part of the text. The up-arrow (↑) or caret (^) is available in all phases of the MILLENNIUM SYSTEMS Assembler and is described in the manner in which it affects macro calls. The example that follows allows the comma and square bracket characters, respectively, to be interpreted as part of the arguments SML,J and [BC] when the macro TIME is invoked:

LABEL	OPERATION	OPERAND	COMMENT
	TIME	1,2,SML^,J,^[BC^]	;INVOKES MACRO TIME AND ;SUBSTITUTES THE ;ARGUMENTS ;1,2,SML,J,AND [BC]

The [ ] Construct (Continued)

The preceding example contains the following arguments:

```
Argument 1 = 1
Argument 2 = 2
Argument 3 = SML,J
Argument 4 = [BC]
```

Additional Macro Argument Conventions

Any leading or trailing blanks are removed from the argument upon macro expansion. Blanks inserted within an argument are retained. If there are only blanks between two commas, the resulting argument is empty. To force a parameter to be replaced by blanks, it may be enclosed within square brackets. Examples of these conventions follow:

```
LABEL     OPERATION     OPERAND

          PQRD          A,B, C ,,[ D,E ]," ",[ ],[^[]
```

The preceding example expands to the following arguments. Asterisks are used only in this example to indicate the beginning and end of the argument and are not expanded as part of the macro text.

```
Argument 1 = *A*
Argument 2 = *B*
Argument 3 = *C*
Argument 4 = **
Argument 5 = * D,E *
Argument 6 = *" "*
Argument 7 = *  *
Argument 8 = *[*
```

Any number or length of arguments may be entered within the operand field of a macro call, as long as the line does not exceed 128 characters (not including a carriage return). In addition, after arguments are substituted for parameters, the lines resulting from the macro expansion must not exceed 128 characters. Otherwise, an error code is displayed.

Examples

The following text includes two examples of macro definition, calling, and the resulting expansions. The first example illustrates a simple macro expansion. The second example is more complex and illustrates two contiguous macro expansions, where one is referenced by the other.

Example 1

In this example, a macro is defined as EVALC. Two parameters, 1 and 2, are defined and surrounded by single quotes within the macro definition block.

LABEL	OPERATION	OPERAND	COMMENT
	MACRO	EVALC	;DEFINES EVALC AS MACRO ;NAME
	BYTE	5,'1'	;ALLOCATES ONE BYTE OF ;MEMORY FOR THE CONSTANT ;VALUE 5 AND ONE BYTE FOR ;THE FIRST PARAMETER ;WITHIN EVALC
	WORD	'2'	;ALLOCATES TWO BYTES OF ;MEMORY FOR THE SECOND ;PARAMETER WITHIN EVALC
	ENDM		;END OF MACRO DEFINITION

Assume the following call appears within a user program.

LABEL	OPERATION	OPERAND	COMMENT
	EVALC	25,357	;INVOKES MACRO EVALC AND ;SUBSTITUTES THE ;ARGUMENTS 25 AND 357 FOR ;THE FIRST TWO ;PARAMETERS WITHIN EVALC

This macro call generates the following macro expansion and substitutes the arguments 25 and 357 for the first two parameters ('1' and '2') within the macro definition block. The argument 357 requires two bytes of memory as defined by the WORD statement within the macro definition block.

LABEL	OPERATION	OPERAND
	BYTE	5,25
	WORD	357

Examples (Continued)

Example 2

In the following example, two macro definition blocks are sequentially defined Q1 and Q2. One parameter is defined within each macro definition block. A macro call, Q1 7, is defined within Q2. This statement calls the macro, Q1.

LABEL	OPERATION	OPERAND	COMMENT
	MACRO	Q1	;DEFINES Q1 AS MACRO NAME
PARM1	ASET	1	;ALLOWS SYMBOLIC REFERENCE ;TO THE FIRST PARAMETER
	BYTE	3,5,'PARM1'	;ALLOCATES ONE BYTE OF ;MEMORY EACH FOR THE ;CONSTANT VALUES 3 AND 5, ;AND FOR THE FIRST ;PARAMETER PASSED TO Q1, ;'PARM1'
	ENDM		;END OF MACRO DEFINITION Q1
	MACRO	Q2	;DEFINES Q2 AS MACRO NAME
	BYTE	3,5,'1'	;ALLOCATES ONE BYTE OF ;MEMORY EACH FOR THE ;CONSTANT VALUES 3 AND 5, ;AND FOR THE FIRST ;PARAMETER PASSED TO ;Q2,'1'
	Q1	7	;CALLS MACRO Q1 AND ;ASSIGNS THE VALUE 7 TO THE ;FIRST PARAMETER PASSED ;TO Q1,'PARM1'
	BYTE	8,9,10	;ALLOCATES ONE BYTE OF ;MEMORY EACH TO THE ;CONSTANT VALUES 8,9, AND ;10
	ENDM		;END OF MACRO DEFINITION :Q2

Assume the following macro call appears within a user program to invoke the macro defined as Q2:

LABEL	OPERATION	OPERAND	COMMENT
	Q2	3	;CALLS THE MACRO Q2 AND ;SUBSTITUTES THE ARGUMENT ;3 FOR THE FIRST PARAMETER ;'1'

Examples (Continued)

This macro call generates the following macro expansion:

LABEL	OPERATION	OPERAND
	BYTE	3,5,3
	BYTE	3,5,7
	BYTE	8,9.10

In this example, the macro call Q2 3, causes the first statement within the macro Q2, BYTE 3,5,'1', to be expanded to BYTE 3,5,3. Expansion proceeds to the next statement that calls the macro Q1 and appears as Q1 7. This statement causes expansion to continue with the statement, PARM1 ASET 1, thus allowing PARM1 to be used as a symbolic reference to the first parameter. This causes the next statement within Q1 to be expanded as BYTE 3,5,7, replacing BYTE 3,5,'PARM1'. Expansion within macro Q1 terminates then terminates with the ENDM directive. This termination causes expansion to continue with the next statement in the referencing macro, Q2. The statement BYTE 8,9,10 is the next statement that is expanded. Control then returns to the main program upon expansion of the ENDM directive, which terminates the macro expansion, Q2.

Conditional Assembly

Macros may be defined such that their expansion is conditional; that is, based upon the values of the parameters they use. IF - ELSE - ENDIF blocks allow conditional assembly and are valid in all phases of the MILLENNIUM SYSTEMS Assembler. REPEAT - ENDR blocks also allow conditional assembly and are only valid within a macro definition. The two methods for performing conditional assembly are summarized as follows. For further information pertainint to IF - ELSE - ENDIF and REPEAT - ENDR usage, refer to Chapter 4, ASSEMBLER DIRECTIVES.

	OPERATION	OPERAND	
1)	IF	expr	Turns off the assembly process if the expression is equal to zero (false). Succeeding statements are passed over and are not acted upon until the ENDIF, or optional ELSE, statement is encountered.
	ELSE		Regenerates assembly process when IF expression equals zero. Usage is optional.
	ENDIF		Terminates the program text controlled by the corresponding IF statement.

(Program continued on next page.)

Conditional Assembly

	OPERATION	OPERAND	
2)	REPEAT	expr1,expr2	If expr 1 is equal to zero (false), statements up to the ENDR statement are ignored. Otherwise, the statements are assembled and the assembler repeats the process again until the expression is equal to zero. A REPEAT block stops iterating when the specified expression maximum, expr2, is reached. If expr2 is not specified, the REPEAT block stops after 255 iterations.
	ENDR		Terminates the program text controlled by the corresponding REPEAT statement.

Nesting

IF - ELSE - ENDIF blocks and REPEAT - ENDR blocks may be nested. The nesting depth is limited only by the amount of memory available to the assembler. Each IF condition must be properly nested, having a matching ENDIF statement that occurs within the scope of that particular IF condition. Only one ELSE directive is permitted within each IF - ENDIF block. In addition, each REPEAT condition must be properly nested, having a matching ENDR statement occurring within the scope of that particular REPEAT condition. IF - ENDIF and REPEAT - ENDR blocks may not cross the boundary of a macro expansion or the boundaries of each other.

Conditional Macro Termination

The EXITM directive terminates the current macro expansion before the assembler encounters an ENDM directive. The EXITM directive is generally used within IF - ELSE - ENDIF and REPEAT - ENDR blocks to conditionally terminate macro expansions. EXITM is valid only within macro definition blocks.

EXAMPLES

IF - ENDIF Blocks

The following example demonstrates the definition, calling, and expansion of a macro using an IF - ENDIF block. The example also demonstrates the use of an EXITM directive to conditionally terminate the macro expansion. In this example, a macro is defined as CONDIF and uses four parameters.

```
LABEL     OPERATION      OPERAND              COMMENT

          MACRO          CONDIF               ;DEFINES CONDIF AS MACRO
                                              ;NAME
          BYTE           '1','2',0,0,0        ;ALLOCATES ONE BYTE OF
                                              ;MEMORY FOR EACH OF FIVE
                                              ;VALUES. THE FIRST AND
                                              ;SECOND VALUES ARE THE
                                              ;FIRST AND SECOND
                                              ;PARAMETERS FOR
                                              ;SUBSTITUTION BY THE MACRO
                                              ;CALL ARGUMENTS. THE 3RD,
                                              ;4TH, AND 5TH VALUES ARE
                                              ;THE CONSTANT, 0
          IF             " '3'"=""            ;TESTS 3RD PARAMETER TO
                                              ;DETERMINE IF IT EXISTS
          BYTE           255                  ;IF 3RD PARAMETER DOES NOT
                                              ;EXIST, ONE BYTE IS
                                              ;GENERATED CONTAINING
                                              ;255 DECIMAL
          EXITM                               ;TERMINATES MACRO
                                              ;EXPANSION, IF CONDITION
                                              ;IS SATISFIED
          ENDIF                               ;END OF IF CONDITION
          BYTE           '3'                  ;OTHERWISE, ONE BYTE IS
                                              ;ASSIGNED CONTAINING 3RD
                                              ;PARAMETER
          BYTE           HI('4'),LO('4')      ;SWAPS BYTES OF 4TH
                                              ;PARAMETER
          ENDM                                ;END OF MACRO DEFINITION
```

IF - ENDIF Blocks (Continued)

Assume the following macro call appears within a main program.

LABEL	OPERATION	OPERAND	COMMENT
	CONDIF	22,29,27,25	;INVOKES MACRO CONDIF AND ;USES THE ARGUMENTS 22,29, ;27, AND 25 FOR SUBSTITUTION ;OF THE FIRST FOUR ;PARAMETERS

This macro call substitutes the arguments 22, 29, 27, and 25 for the parameters labeled '1', '2', '3', and '4'. Notice that the substitution indicator (+) is displayed prior to each listed source line where substitution occurs.

```
0000  161D0000+      BYTE     22,29,0,0,0         ;ALLOCATES ONE BYTE OF
                                                   ;MEMORY

0004  00
0005  1B       +     BYTE     27                  ;OTHERWISE, ONE BYTE IS
                                                   ;ASSIGNED
0006  0019     +     BYTE     HI(25)LO(25)        ;SWAPS BYTES OF 4TH
                                                   ;PARAMETER
```

If the third substituted argument in this expansion had been empty rather than 27, the EXITM statement would have terminatedf further macro expansion.

REPEAT - ENDR Blocks

In the following example of a REPEAT - ENDR block, a macro is defined as CONDR and defines the ASET symbol, AGAIN.

LABEL	OPERATION	OPERAND	COMMENT
	MACRO	CONDR	;DEFINES CONDR AS MACRO ;NAME
AGAIN	ASET	1	;INITILIAZES AGAIN TO EQUAL ;1 AT ASSEMBLY TIME
	REPEAT	AGAIN < + '#'	;REPEAT WHILE AGAIN IS LESS ;THAN OR EQUAL TO TOTAL ;NO. OF ARGUMENTS ON THIS ;CALL
	BYTE	'AGAIN'	;GENERATES ONE BYTE OF ;MEMORY CONTAINING THE ;CURRENT PARAMETER
AGAIN	ASET	AGAIN + 1	;INCREMENT AGAIN AT ;ASSEMBLY TIME
	ENDR		;END OF REPEAT CONDITION
	BYTE	ODH	;GENERATES A CARRIAGE ;RETURN
	ENDM		;END OF MACRO DEFINITION

Assume the following macro call appears within a main program.

LABEL	OPERATION	OPERAND	COMMENT
	CONDR	25,26,27	;INVOKES MACRO CONDR AND ;SUBSTITUTES THE ARGUMENTS ;25, 26, AND 27 FOR THE FIRST ;THREE PARAMETERS

REPEAT - ENDR Blocks (Continued)

This macro call generates the following macro expansion and substitutes the arguments 25, 26, and 27 for the parameter labeled 'AGAIN'. The substitutions occur for as many times as there are arguments specified in the macro call, as defined by the '#' character. In this case, there are three arguments specified and the '#' character is replaced by 3.

```
                    0001     AGAIN     ASET      1
                    FFFF     +         REPEAT    AGAIN<=00003
             0000   19       +         BYTE      25
                    0002     AGAIN     ASET      AGAIN+1
                                       ENDR
                    FFFF     +         REPEAT    AGAIN<=00003
             0001   1A       +         BYTE      26
                    0003     AGAIN     ASET      AGAIN+1
                                       ENDR
                    FFFF     +         REPEAT    AGAIN<=00003
             00002  1B       +         BYTE      27
                    0004     AGAIN     ASET      AGAIN+1
                                       ENDR
             0003   0D                 BYTE      0DH
                                       ENDM
   00005     0004            END
```

Macro Expansion Summary

The lines of code within the macro definition block are not assembled with the rest of the program, but are saved until macro expansion time. Blank lines or comment lines are exceptions to this rule since they are not saved for expansion. The macro definition block, therefore, does not generate object code upon assembly. When the macro name appears within the operation field of the main program during assembly, the body of the macro is inserted and assembled within the main program.

Prior to the assembly of each line in the macro definition block, the assembler scans for the presence of the single quote character. An argument defined in the macro call then replaces the parameter within the single quote characters. After substitution, the scan continues from the first character following the replaced text until the end of the current line. The line is inserted into the user program. The assembler then generates object code and processes the line. The assembler continues to obtain lines from the macro definition block in this manner until an ENDM or EXITM statement is encountered. At that time, expansion continues with the statement following the macro call.

ASSEMBLER OPERATING PROCEDURES

INTRODUCTION

This chapter describes the syntax required for the MILLENNIUM SYSTEMS Assembler to translate source code into executable binary object code.

```
SYNTAX

        {ASM}{Filename} [F1 F2 F3]
```

PURPOSE

The ASM (assembler) command allows the user to assemble a specified program on a disk.

EXPLANATION

The command ASM means assemble. The filename is the name of the source file to be assembled. The file type extension is not included in the command. MP/M or CP/M uses the file name to generate:

- The source Filename by appending .ASM
- The list Filename by appending .PRN
- The object Filename by appending .HEX

The three flags (F1, F2, F3) are optional. The first flag is associated with the source file, the second flag is the object file, and the third flag with the list file. The flags are one character each and have the following meaning:

A through P - Logical disk drives
Z - Do not produce a file
X - Applies only to Flag 3 (F3) and means put the Listing on the console.

NOTE: If the filename has a logical drive designator (A:filename) and the flag specifies a different logical drive, the flag takes precedence.

INTRODUCTION

The assembler listing is composed of two parts:

1) the source program assembler listing with the object code generated for each instruction; and

2) a table of all symbols used in the program.

THE ASSEMBLER LISTING

The assembler listing is composed of headings, lines of source code listing information, and error responses relating to any assembling errors.

Headings

Each page of the assembler listing contains a heading. The heading includes the assembler version on the left side of the page, and the page number on the right side of the page, as shown below:

```
MILLENNIUM Z80 ASM VX.X                                          PAGE X
```

If the TITLE directive is used, a 30-character string expression may be inserted at the top of each listing page for program identification. The character string specified as the TITLE operand is printed on the first character line between the assembler version number and the page number, shown as follows:

```
MILLENNIUM Z80 ASM VX.S  THIS IS THE PROGRAM TITLE               PAGE X
```

If the STITLE directive is used, a 72-character string expression may be inserted on the second line of each listing page for program identification. The character string specified as the STITLE operand is printed between the page heading and the first source code line. A blank line is automatically inserted between the string and the beginning of the source code. A program identification heading created with the STITLE directive appears below:

```
MILLENNIUM Z80 ASM VX.X                                          PAGE X
THIS LINE DEMONSTRATES STITLE USAGE
(blank line)
.
.  (source code)
.
```

The Listing Line

The heading is followed by a blank line and the listing information. Each source program line is translated and output in the following sequence:

1) a line number,

2) the memory location of the instruction or data,

3) the translated object code,

4) a relocation indicator if relocation occurs on the line,

5) a substitution indicator if substitution occurs on the line, and

6) the original source line.

The listing line may be 72 or 132 characters wide, dependent upon whether the TRM option for the LIST and NOLIST directives is active. The first listing line field is a five-character decimal line number. Line numbers are not listed for macro expansion lines. The second listing field is a four-character hexadecimal location counter. This field may also represent a symbol value for an EQU directive. Both the line number and the location counter are right justified with leading zeros when necessary, and are separated from each other by one space.

The object field follows the location counter field, and the fields are separated by one space. The object code is left justified and may be a maximum of eight hexadecimal characters wide. If an instruction generates more than eight hexadecimal characters, all additional object code is listed on subsequent lines.

If relocation occurs in a line, the greater-than character (>) follows the object field. Actual relocation is performed at link time.

If a substitution occurs in a line, the plus character (+) follows the relocation indicator or the object field. All substitutions occur before the line is listed. The example that follows shows the plus sign preceding a line where a substitution occurs.

```
00001 0000 030502 + BYTE 3,5,2                    ;ALLOCATE ONE BYTE OF
                                                  ;MEMORY FOR EACH OF THE
                                                  ;CONSTANT VALUES 3 AND 5,
                                                  ;AND FOR THE VALUE DEFINED
                                                  ;TO SUBSTITUTE FOR '1' (IN
                                                  ;THIS CASE THE VALUE IS 2)
```

The Listing Line (Continued)

The source code follows the relocation or substitution indicators or the object code field, and the fields are separated by one space. If the TRM option is ON when entered with the LIST directive, 52 spaces remain in the listing line for the source code. Any source code exceeding the 52-character limit is truncated. If the TRM option is OFF, either by default or when entered with the NOLIST directive, 112 characters remain in the listing for the source code. Any source code exceeding the 112-character limit is truncated.

Any non-printing character, other than the space, tab, or carriage return characters, is represented by a question mark (?) in the listing. The assembler translates the character replaced by the ? to the original character form.

To summarize, the listing line appears as follows:

```
XXXX  LLLL  D D D D D D D D > + SSSSS.....
```

Each field is represented as follows:

X = Line number, right justified
L = Memory location (or EQU statement symbol value)
D = Object code
> = Relocation indicator (relocation is performed at link time)
+ = Substitution indicator (substitution has occurred before listing)
S = Source line

Error REsponse

If an error occurs in an instruction, the line containing the error is followed by an error response. This is also true when the instruction generates more than one line of object code. The error response takes the following form:

```
*****ERROR code
```

The "code" in the above error response is replaced by a three-digit number indicating the type of error detected. For a description of all error codes and their corresponding messages, refer to Appendix D. If the error response precedes an additional message, "FATAL ERROR; ASSEMBLY ABORTED AT LINE XXXX", the severity of the error is such that the Assembler cannot continue execution.

THE SYMBOL TABLE

The symbol table follows the listing, indicating all symbols used in the source module and the values these symbols represent. The symbol table also categorizes all symbols according to their type or base, for ease in referencing. The structure of the symbol table follows a three-part format: a heading, symbols and their values (categorized by type or base), and two lines providing statistical program assembly information.

Each symbol table page contains a heading following the format shown below:

```
MILLENNIUM Z80 ASM Vx.x      SYMBOL TABLE LISTING                              PAGE x
```

Below the heading, symbols and their corresponding hexadecimal values appear in categories according to their type or base. Headings precede each category describing the group of symbols in each category. The possible symbol headings are as follows:

STRING AND MACROS	All string and macro symbols are listed under this category.
SCALARS	All symbols having scalar values and all undefined symbols are listed under this category.
name SECTION characteristic (length)	All symbols based to the named Linker section are listed. The section characteristic indicates whether the section is relocated at the starting address of a physical memory block (PAGE), whether the section is relocated on any byte address within a page (INPAGE), or whether the section is based to the actual address specified by the ORG directive at assembly time (ABSOLUTE). Refer to the discussion on Section Definition Directives in Chapter 4. If no characteristic is listed, the section is byte relocatable. The length of the named section is specified in bytes.
name COMMON characteristic (length)	Same as SECTION category, except that more than one common section with the same name is valid at link time.

THE SYMBOL TABLE

name RESERVE characteristic (length)	Same as SECTION category, except that all sections with the specified name are combined into a single section at link time.
name UNBOUND GLOBAL	An unbound global is a symbol declared in a global statement, having no value in this assembly. The named unbound global must be defined in other assemblies or at link time. If an unbound global is used to assign a value to a symbol in this assembly, that symbol is listed under the UNBOUND GLOBAL category in the symbol table listing.

Columns containing symbols and their corresponding hexadecimal values are listed alphabetically under each category. When a symbol has fewer than eight characters, dashes and spaces (- - -) serve as padding between a symbol and its value. The value field contains four hexadecimal characters and right justified, with leading zeros where necessary. The value field for undefined symbols appears as a series of asterisks (****). Each value is followed by several spaces and the next symbol. A typical symbol table listing line might appear as follows:

```
SYM --- 0101   SYMB2 - - 0025  SYMB3 - -0022   SYMBOL4 **** SYMBOL5 0121
```

The number values for string and macro symbols indicate the number of bytes used by the symbol for text storage. The number values for ASET symbols indicate the last values assigned to the symbols. The number values for GLOBAL and ENDOF symbols represent the addresses prior to relocation.

Symbol indicators may appear after the symbol values. An indicator also appears if a high or low truncation occurs at link time. The symbol indicators are summarized as follows:

- S - String symbol
- M - Macro symbol
- V - ASET symbol
- G - Global symbol
- H - High truncation indicator (truncation will occur at link time)
- L - Low truncation indicator (truncation will occur at link time)
- E - ENDOF symbol (value will be adjusted at link time)

All symbols without indicators are EQU symbols. The number values for these symbols indicate their values during assembly.

THE SYMBOL TABLE

If the TRM option is specified with the NOLIST directive, or is otherwise OFF due to default, the symbol table listing is five columns wide. If the TRM option is specified with the LIST directive, causing the option to be ON, the symbol table listing is three columns wide.

Two lines appear below the symbol table display providing statistical information about the current assembly. The first line shows the number of source lines, the number of assembled lines, and the number of available bytes. The number of available bytes indicates the amount of space available for further data manipulation or symbol storage within the assembler. The second statistical line indicates the number of errors and undefined symbols, if any.

A sample assembler and symbol table listing is shown in figure 7-1.

```
MILLENNIUM Z80 ASM V3.3     THIS IS THE TITLE                              PAGE 1
THIS LINE IS THE STITLE OF MY PROGRAM

00003                            STRING  S1(80)    ;DEFINE STRING VARIABLE S1 WITH
                                                   ;80-CHARACTER MAXIMUM
00004              0003  L1      EQU     3         ;DEFINE CONSTANT SYMBOL L1 TO
***** ERROR        003                             ;EQUAL 3

00005              0004  L2      ASET    4         ;DEFINE VARIABLE SYMBOL L2 TO
                                                   ;EQUAL 4
00006              0100>         ORG     100H      ;STARTS OBJECT CODE OF NEXT
                                                   ;INSTRUCTION AT 100H
00007  0100 7E           L1      LD      A,(HL)    ;LOAD REG. A WITH CONTENTS OF
***** ERROR        002                             ;MEMORY POINTED TO BY HL
                                                   ;REGISTER PAIR.  MULTIPLY-DEFINED
                                                   ;SYMBOL, L1.

00008                            END               ;END OF PROGRAM
.
.
.
.
MILLENNIUM Z80 ASM V3.3     SYMBOL TABLE LISTING                           PAGE 2


STRINGS AND MACROS

        S1 - - - - - 0050 S

SCALARS

        L2 - - - - - 0004 V

% (default) SECTION (0101)

        L1 - - - - - 0100
15 SOURCE LINES          15 ASSEMBLED LINES          1000 BYTES AVAILABLE
2 ERRORS
```

Figure 7-1. Sample Assembler and Symbol Table Listing

SOURCE MODULE CHARACTER SET

SYMBOLS	DEFINITION
A..Z	letters used in symbols; lower-case characters (other than in strings and comments) are interpreted as the corresponding upper-case characters
0...9	numbers used in symbols and constants
$	used in symbols, and to represent assembler location counter contents
.	used in symbols
-	used in symbols
;	precedes a comment
,(comma)	delimiter for operand items
"	string delimiter
:	string concatenation operator
'	string substitution delimiter
#	total number of arguments passed to current macro expansion
[ ]	group macro code to be treated as a single argument
@	provides unique labels for each macro expansion
%	is replaced by name of current section or common in a macro expansion
*	binary arithmetic operation, multiplication
/	binary arithmetic operation, division
+	unary or binary arithmetic operator, addition
-	unary or binary arithmetic operator, subtraction
( )	override precedence of operators
\	unary logical operator, not

SOURCE MODULE CHARACTER SET

SYMBOLS	DEFINITION
&	binary logical operator, and
!	binary logical operator, inclusive or
!!	binary logical operator, exclusive or
SPACE	field delimiter
TAB	field delimiter
CARRIAGE RETURN	field and line delimiter
∧ or ↑	allows following special character to have literal meaning
∧∧ or ↑↑	allows the second caret or up-arrow character to have literal meaning
=	relational operator, equal
< >	relational operator, not equal
>	relational operator, greater than
<	relational operator, less than
>=	relational operator, greater than or equal
<=	relational operator, less than or equal

ASSEMBLER DIRECTIVES

DIRECTIVE	OPERATION
ASCII	stores ASCII text in memory
ASET	assigns or reassigns an expression value to a string or numeric variable symbol
BLOCK	reserves a specified number of bytes in memory FF!!
BYTE	allocates one byte of memory to each expression specified
COMMON	declares Linker section, assigns name, defines type to be common
ELSE	when expression is false, causes assembly of alternate source lines between ELSE and ENDIF directives
END	terminates source modules
ENDIF	signals corresponding IF block termination
ENDM	terminates a macro definition block
ENDR	signals end of each REPEAT cycle
EQU	permanently assigns a value to a symbol
EXITM	terminates expansion of current macro before encountering ENDM
GLOBAL	declares symbols to be global variables
IF	when expression is true, causes assembly of source lines between IF and ENDIF directives
INCLUDE	inserts text from specified file into the program
LIST	enables display of assembler listing features
MACRO	defines the name of a source code block used repeatedly within a program
NAME	declares name of an object module
NOLIST	disables display of assembler listing features
ORG	sets contents of location counter

DIRECTIVE	OPERATION
PAGE	begins the next listing line on the following page
REPEAT	enables macro lines between REPEAT and ENDR directives to be assembled repeatedly
RESERVE	sets aside a workspace in memory
RESUME	continues definition of code for a given section
SECTION	declares Linker section, assigns name, defines parameters
SPACE	spaces downward a specified number of listing lines
STITLE	creates a text line on the second line of each listing page heading for program identification
STRING	declares symbol to be a string variable
TITLE	creates a text line at the top of each listing page heading for program identification
WARNING	generates specified warning message on the output device and in the listing
WORD	allocates two bytes of memory to each expression specified

ASSEMBLER DIRECTIVE SYNTAX

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	ASCII	{string expression} [,string expression]...	[;charstring]
{symbol}	ASET	{expression}	[;charstring]
[symbol]	BLOCK	{expression}	[;charstring]
[symbol]	BYTE	{expression} [,expression]...	[;charstring]
[symbol]	COMMON	{symbol} [,PAGE / ,INPAGE / ,ABSOLUTE]	[;charstring]
[symbol]	ELSE		[;charstring]
[symbol]	END	[expression]	[;charstring]
[symbol]	ENDIF		[;charstring]
[symbol]	ENDM		[;charstring]
[symbol]	ENDR		[;charstring]
{symbol}	EQU	{expression}	[;charstring]
[symbol]	EXITM		[;charstring]
[symbol]	GLOBAL	{symbol} [,symbol]...	[;charstring]
[symbol]	IF	{expression}	[;charstring]
[symbol]	INCLUDE	{string expression}	[;charstring]
[symbol]	LIST	[CND] [,TRM] [,SYM] [,CON] [,MEG] [,ME]	[;charstring]
[symbol]	MACRO	{symbol}	[;charstring]
[symbol]	NAME	{symbol}	[;charstring]
[symbol]	NOLIST	[CND] [,TRM] [,SYM] [,CON] [,MEG] [,ME]	[;charstring]
[symbol]	ORG	{[/]expression}	[;charstring]
[symbol]	PAGE		[;charstring]

(Directives continued on next page)

ASSEMBLER DIRECTIVE SYNTAX (Continued)

LABEL	OPERATION	OPERAND	COMMENT
[symbol]	REPEAT	{expression1} [,expression2]...	[;charstring]
[symbol]	RESERVE	{symbol,expression} [,PAGE / ,INPAGE]	[;charstring]
[symbol]	RESUME	[symbol]	[;charstring]
[symbol]	SECTION	{symbol} [,PAGE / ,INPAGE / ,ABSOLUTE]	[;charstring]
[symbol]	SPACE	[expression]	[;charstring]
[symbol]	STITLE	{string expression}	[;charstring]
[symbol]	STRING	{{strvar1} [(lenexp1)]} [,strvar2 [(lenexp2)]] ...	[;charstring]
[symbol]	TITLE	{string expression}	[;charstring]
[symbol]	WARNING		[message]
[symbol]	WORD	{expression}[,expression]...	[;charstring]

Appendix C

HEXADECIMAL CONVERSION TABLES

ASCII CODE CONVERSION TABLE

HEXADECIMAL

LEAST SIGNIFICANT CHARACTER (rows) / MOST SIGNIFICANT CHARACTER (columns)

—	Ø	1	2	3	4	5	6	7
Ø	NUL	DLE	SP	Ø	@	P	`	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	¦
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	∧	n	~
F	SI	US	/	?	O	—	o	DEL

EXAMPLES

W = 57
H = 48
a = 61
t = 74
@ = 4Ø
NUL = ØØ
DEL = 7F

Decimal-Hexadecimal-Binary Equivalents 0-255$_{10}$

Deci-mal	Hexa-deci-mal	Binary 8-bit Code	Deci-mal	Hexa-deci-mal	Binary 8-bit Code	Deci-mal	Hexa-deci-mal	Binary 8-bit Code	Deci-mal	Hexa-deci-mal	Binary 8-bit Code
0	00	0000 0000	64	40	0100 0000	128	80	1000 0000	192	C0	1100 0000
1	01	0000 0001	65	41	0100 0001	129	81	1000 0001	193	C1	1100 0001
2	02	0000 0010	66	42	0100 0010	130	82	1000 0010	194	C2	1100 0010
3	03	0000 0011	67	43	0100 0011	131	83	1000 0011	195	C3	1100 0011
4	04	0000 0100	68	44	0100 0100	132	84	1000 0100	196	C4	1100 0100
5	05	0000 0101	69	45	0100 0101	133	85	1000 0101	197	C5	1100 0101
6	06	0000 0110	70	46	0100 0110	134	86	1000 0110	198	C6	1100 0110
7	07	0000 0111	71	47	0100 0111	135	87	1000 0111	199	C7	1100 0111
8	08	0000 1000	72	48	0100 1000	136	88	1000 1000	200	C8	1100 1000
9	09	0000 1001	73	49	0100 1001	137	89	1000 1001	201	C9	1100 1001
10	0A	0000 1010	74	4A	0100 1010	138	8A	1000 1010	202	CA	1100 1010
11	0B	0000 1011	75	4B	0100 1011	139	8B	1000 1011	203	CB	1100 1011
12	0C	0000 1100	76	4C	0100 1100	140	8C	1000 1100	204	CC	1100 1100
13	0D	0000 1101	77	4D	0100 1101	141	8D	1000 1101	205	CD	1100 1101
14	0E	0000 1110	78	4E	0100 1110	142	8E	1000 1110	206	CE	1100 1110
15	0F	0000 1111	79	4F	0100 1111	143	8F	1000 1111	207	CF	1100 1111
16	10	0001 0000	80	50	0101 0000	144	90	1001 0000	208	D0	1101 0000
17	11	0001 0001	81	51	0101 0001	145	91	1001 0001	209	D1	1101 0001
18	12	0001 0010	82	52	0101 0010	146	92	1001 0010	210	D2	1101 0010
19	13	0001 0011	83	53	0101 0011	147	93	1001 0011	211	D3	1101 0011
20	14	0001 0100	84	54	0101 0100	148	94	1001 0100	212	D4	1101 0100
21	15	0001 0101	85	55	0101 0101	149	95	1001 0101	213	D5	1101 0101
22	16	0001 0110	86	56	0101 0110	150	96	1001 0110	214	D6	1101 0110
23	17	0001 0111	87	57	0101 0111	151	97	1001 0111	215	D7	1101 0111
24	18	0001 1000	88	58	0101 1000	152	98	1001 1000	216	D8	1101 1000
25	19	0001 1001	89	59	0101 1001	153	99	1001 1001	217	D9	1101 1001
26	1A	0001 1010	90	5A	0101 1010	154	9A	1001 1010	218	DA	1101 1010
27	1B	0001 1011	91	5B	0101 1011	155	9B	1001 1011	219	DB	1101 1011
28	1C	0001 1100	92	5C	0101 1100	156	9C	1001 1100	220	DC	1101 1100
29	1D	0001 1101	93	5D	0101 1101	157	9D	1001 1101	221	DD	1101 1101
30	1E	0001 1110	94	5E	0101 1110	158	9E	1001 1110	222	DE	1101 1110
31	1F	0001 1111	95	5F	0101 1111	159	9F	1001 1111	223	DF	1101 1111
32	20	0010 0000	96	60	0110 0000	160	A0	1010 0000	224	E0	1110 0000
33	21	0010 0001	97	61	0110 0001	161	A1	1010 0001	225	E1	1110 0001
34	22	0010 0010	98	62	0110 0010	162	A2	1010 0010	226	E2	1110 0010
35	23	0010 0011	99	63	0110 0011	163	A3	1010 0011	227	E3	1110 0011
36	24	0010 0100	100	64	0110 0100	164	A4	1010 0100	228	E4	1110 0100
37	25	0010 0101	101	65	0110 0101	165	A5	1010 0101	229	E5	1110 0101
38	26	0010 0110	102	66	0110 0110	166	A6	1010 0110	230	E6	1110 0110
39	27	0010 0111	103	67	0110 0111	167	A7	1010 0111	231	E7	1110 0111
40	28	0010 1000	104	68	0110 1000	168	A8	1010 1000	232	E8	1110 1000
41	29	0010 1001	105	69	0110 1001	169	A9	1010 1001	233	E9	1110 1001
42	2A	0010 1010	106	6A	0110 1010	170	AA	1010 1010	234	EA	1110 1010
43	2B	0010 1011	107	6B	0110 1011	171	AB	1010 1011	235	EB	1110 1011
44	2C	0010 1100	108	6C	0110 1100	172	AC	1010 1100	236	EC	1110 1100
45	2D	0010 1101	109	6D	0110 1101	173	AD	1010 1101	237	ED	1110 1101
46	2E	0010 1110	110	6E	0110 1110	174	AE	1010 1110	238	EE	1110 1110
47	2F	0010 1111	111	6F	0110 1111	175	AF	1010 1111	239	EF	1110 1111
48	30	0011 0000	112	70	0111 0000	176	B0	1011 0000	240	F0	1111 0000
49	31	0011 0001	113	71	0111 0001	177	B1	1011 0001	241	F1	1111 0001
50	32	0011 0010	114	72	0111 0010	178	B2	1011 0010	242	F2	1111 0010
51	33	0011 0011	115	73	0111 0011	179	B3	1011 0011	243	F3	1111 0011
52	34	0011 0100	116	74	0111 0100	180	B4	1011 0100	244	F4	1111 0100
53	35	0011 0101	117	75	0111 0101	181	B5	1011 0101	245	F5	1111 0101
54	36	0011 0110	118	76	0111 0110	182	B6	1011 0110	246	F6	1111 0110
55	37	0011 0111	119	77	0111 0111	183	B7	1011 0111	247	F7	1111 0111
56	38	0011 1000	120	78	0111 1000	184	B8	1011 1000	248	F8	1111 1000
57	39	0011 1001	121	79	0111 1001	185	B9	1011 1001	249	F9	1111 1001
58	3A	0011 1010	122	7A	0111 1010	186	BA	1011 1010	250	FA	1111 1010
59	3B	0011 1011	123	7B	0111 1011	187	BB	1011 1011	251	FB	1111 1011
60	3C	0011 1100	124	7C	0111 1100	188	BC	1011 1100	252	FC	1111 1100
61	3D	0011 1101	125	7D	0111 1101	189	BD	1011 1101	253	FD	1111 1101
62	3E	0011 1110	126	7E	0111 1110	190	BE	1011 1110	254	FE	1111 1110
63	3F	0011 1111	127	7F	0111 1111	191	BF	1011 1111	255	FF	1111 1111

HEX ADDITION

HEXADECIMAL ADDITION TABLE

	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18
A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B
D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

HEX	F + 8	=	17	
HEX	10	=	16	DEC
HEX	7	=	7	DEC
HEX	17	=	23	DEC

HEX MULTIPLY

HEXIDECIMAL MULTIPLICATION TABLE

	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	2	4	6	8	A	C	E	10	12	14	16	18	1A	1C	1E
3	3	6	9	C	F	12	15	18	1B	1E	21	24	27	2A	2D
4	4	8	C	10	14	18	1C	20	24	28	2C	30	34	38	3C
5	5	A	F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
6	6	C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
7	7	E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
8	8	10	18	20	28	30	38	40	48	50	58	60	68	70	78
9	9	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	A	14	1E	28	32	3C	46	50	5A	64	6E	78	82	8C	96
B	B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C	C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D	D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F	F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4	C3	D2	E1

HEX	9 x 8	=	48	
HEX	40	=	64	DEC
HEX	8	=	8	DEC
HEX	48	=	72	DEC

ASSEMBLER ERROR CODES

The following error code numbers signify the MILLENNIUM SYSTEMS Assembler error messages describing them. Upon assembly and in assembler listings, error codes and messages appear immediately below the source line containing an error.

***** ERROR 001: (no message displayed.)

Indicates that a user-entered WARNING message has assembled. Refer to WARNING directive explanation in Chapter 4.

*****ERROR 002: Symbol already defined

Indicates that the symbol defined has been previously defined in the program assembling sequence. Occurs when the same symbol is equated to two values (with EQU directive) or when the same symbol labels two instructions.

*****ERROR 003: Symbol value Phase Error

Indicates that the label or EQU symbol value differs between passes, or that the section aassignment of a label or EQU symbol value differs between passes.

*****ERROR 004: Illegal EQU of GLOBAL

Indicates that an unbound global is assigned the value of another unbound global (with EQU directive). Error occurs because unbound globals are not assigned values in the current assembly.

*****ERROR 005: Global definition may not use HI, LO, or ENDOF

Indicates that the value assigned to the global symbol involved HI, LO, or ENDOF function usage. Occurs when a global symbol is equated to HI(x) or LO(x), where x is an address, or ENDOF(y), where y is the section name whose ending address is to be found.

*****ERROR 006: String expression required

Indicates that a numeric value appears where a string value is required. Operations requiring string expressions involve concatenation, SEG and NCHR function usage, and ASCII, TITLE, or STITLE directive usage.

***** ERROR 007: Undefined BLOCK or ORG expression

The operand expression of an ORG or BLOCK directive is either undefined or a forward reference. Occurs when an undefined or misspelled symbol appears in an ORG or BLOCK directive, or a symbol is assigned a value after the ORG or BLOCK references the symbol.

*****ERROR 008: Invalid ORG out of section

Indicates that the ORG operand expression represents an address defined outside the current section. Examine previous RESUME or SECTION statements for errors.

*****ERROR 009: Negative block length

Indicates that the BLOCK operand expression represents a negative value.

*****ERROR 010: Macro already defined

Indicates that more than one MACRO directive contains the same name.

*****ERROR 011: Macro definition phase error

Indicates two possible errors: The macro was called before being defined, or the macro was defined during the second assembler pass, but not the first.

*****ERROR 012: Memory full on Macro call

Indicates insufficient space to perform macro expansion. Occurs when too many long arguments are specified for parameter substitution, too many symbols are entered in macro definition, or the macro repeats itself infinitely.

*****ERROR 013: Missing ENDR or ENDIF

Indicates that a conditional assembly (IF or REPEAT) block failed to complete assembly. Occurs when a conditional assembly block begins assembly within a macro definition and the macro terminates (with an ENDM directive) before the conditional assembly terminates (with an ENDR or ENDIF directive).

***** ERROR 014: Duplicate definition of section name

Indicates that the section name has already been defined as a label symbol during the current assembler pass.

*****ERROR 015: END directive invalid within an INCLUDE file

Indicates an invalid END directive is present within an INCLUDE file.

*****ERROR 016: ENDR or ENDIF mis-matched

Indicates that an improper termination directive was used for a conditional assembly block. Occurs when ENDR is entered to terminate an IF block, ENDIF is entered to terminate REPEAT block, or when IF and REPEAT blocks overlap each other producing the same effect.

*****ERROR 017: Iteration limit exceeded

Indicates an attempt to assemble a REPEAT block more than the specified number of times. If the allowed number of repeat cycles is left unspecified, the error message is displayed when 256 repeat cycles are completed.

*****ERROR 018: Misplaced ELSE

Indicates that an ELSE directive occurs outside its corresponding IF-ENDIF block, or that more than one ELSE directive occurs within the scope of one IF-ENDIF block.

*****ERROR 019: Operation invalid for address

Indicates that an operation allowing only scalar values was applied to an address value.

*****ERROR 020: Divisor is zero

Indicates that the Assembler attempted to divide by zero. Also occurs when the Assembler attempts to determine the remainder of a division by zero with the MOD operator (for example, A MOD 0).

***** ERROR 021: Text following] ignored

Indicates that information following a bracketed macro parameter has been ignored.

*****ERROR 022: ENDOF operand is scalar

Indicates that the specified section name in the ENDOF statement is a non-global, scalar symbol.

*****ERROR 023: ENDOF already applied

Indicates an attempt to perform an ENDOF function upon an address resulting from a previous ENDOF function.

*****ERROR 024: ENDOF operand is not global

Indicates that the specified section name in the ENDOF statement represents a non-global symbol.

*****ERROR 025: Operation on HI or LO of address

Indicates an attempt to perform an arithmetic or unary operation upon an address that has had HI or LO applied to it.

*****ERROR 026: Addition of addresses

Indicates an attempt to add one address to another.

*****ERROR 027: Conflicting section bases

Indicates an attempt to subtract or compare addresses based to different sections or having different ending byte addresses.

*****ERROR 028: Address subtracted from scalar

Indicates an attempt to subtract an address from a scalar value.

*****ERROR 029: Negative string length

Indicates that a negative value was specified for the string length when the string was declared with the STRING directive.

*****ERROR 030: String length phase error

Indicates that the string expression value differs between the assembler's first and second pass. Occurs when the string length expression contains a forward reference.

*****ERROR 031: Redeclaration of string variable

Indicates a second attempt to declare the same string variable.

*****ERROR 032: String declaration phase error

Indicates that the string value was defined during the assembler's second pass, but not its first.

*****ERROR 033: Invalid string name

Indicates that an invalid string variable name has been entered as an operand in the STRING directive.

*****ERROR 034: END inside an unclosed block

Indicates that an END statement occurs within an IF, REPEAT, or MACRO definition block. Occurs when an ENDIF, ENDR, or ENDM directive is either missing or misspelled.

*****ERROR 035: Value truncated to byte

Indicates that the value entered exceeds one byte (value falls outside the range-128 to 255). The value is truncated to fall within one-byte range.

*****ERROR 036: Invalid character follows label

Indicates that a character other than a space was encountered following a label.

***** ERROR 037: Extra operans ignored

Indicates that extra operands appear in the statement. The complete statement entered prior to the extra operands is assembled, and the extra operands are ignored. Occurs when a statement is miscoded, an invalid delimiter occurs in the operand list, or a semicolon does not precede a comment. This error also occurs when a logical not "\" operator or a function follows what could be interpreted as a complete expression. This complete expression is either composed of or ends in a constant, a symbol, or a right parentheses ")". The portion of the statement that precedes the logical not operator or function is assembled and the remaining portion of the operand is ignored.

*****ERROR 038: String variable used as label

Indicates that a string variable is present in the label field of an instruction. Label is ignored.

*****ERROR 039: Invalid operation code

Indicates that the Assembler is unable to recognize the operation in the statement, or that the Assembler disallows the operation to be processed in its entered context. Occurs when the operation is misspelled, an invalid delimiter follows the label, or a macro is called prior to its definition.

*****ERROR 040: Invalid character

Indicates that the Assembler has encountered a character, outside the valid character set, that was not enclosed within double quotes.

*****ERROR 041: Syntax error

Indicates that the statement does not conform to the required syntax. Refer to Appendix B for required syntax for Assembler directives.

*****ERROR 042: Invalid option or separator

Indicates that the Assembler encountered an invalid delimiter between listing control options in the LIST or NOLIST directive operand field. Occurs when spaces delimit the options where commas are required, or when an invalid listing control option is entered.

*****ERROR 043: No label on EQU or ASET

Indicates that a symbol is either missing from or invalid for the label field of an EQU or ASET directive.

*****ERROR 044: Invalid Macro name

Indicates that the macro name is missing from the operand field of the MACRO directive, or that the macro name is an invalid symbol. Occurs when a previously-defined symbol is entered as a macro name, a macro name is missing from the macro directive operand field, or an invalid delimiter is entered between the macro operation and macro name.

*****ERROR 045: Invalid relocation option

Indicates an attempt to specify an invalid relocation option (other than PAGE, INPAGE, or ABSOLUTE) when declaring a section. When this error occurs, the assembler ignores the invalid option, and handles the specified section as if it were byte relocatable.

*****ERROR 046: MACRO within a Macro

Indicates that a macro definition statement was encountered within a macro expansion or a macro definition block.

*****ERROR 047: Invalid except in Macro

Indicates that an EXITM, ENDM, REPEAT, or ENDR directive appeared outside a macro definition block.

*****ERROR 048: Invalid operand

Indicates that the specified operand is either incomplete or inaccurate for the context required by the operation. If the required operand is an expression, this error indicates that the first item in the operand field is not a variable, constant, a left parentheses "(", a minus sign "-", or a logical not "\".

*****ERROR 049: Address assigned to string

Indicates an attempt to assign an address value to a string variable symbol.

*****ERROR 050: Section definition Phase error

Indicates that the specified section or relocation option differs between the Assembler's first and second pass.

*****ERROR 051: Section definition Phase error

Indicates that the specified section was defined during the second, but not the first, Assembler pass.

*****ERROR 052: Too many Section or Globals

Indicates that the number of declared sections and global symbols exceeds 254. The Assembler does not accept the current section or global declaration.

*****ERROR 053: Invalid relocation option

Indicates that the ABSOLUTE relocation option was specified in the RESERVE directive operand field.

*****ERROR 054: Negative RESERVE length

Indicates that a negative-valued byte length was specified as the RESERVE operand expression.

*****ERROR 055: Invalid section name

Indicates that an invalid symbol was declared as a SECTION, COMMON, or RESERVE name. Occurs when the symbol name is misspelled, contains invalid characters, is a reserved word, or is a previously-defined label.

*****ERROR 056: Invalid RESERVE length

Indicates that the required RESERVE operand expression (specifying the number of bytes reserved for the current object module) is either entered incorrectly, entered without a comma before the expression, or absent from the RESERVE directive.

*****ERROR 057: RESUME section undefined

Indicates that the resumed section is defined in a later statement in the assembly process.

*****ERROR 058: RESUME of RESERVE section

Indicates an attempt to resume a reserved section.

*****ERROR 059: Resumed section invalid

Indicates that the resumed section was declared after the 254th section or global symbol was declared.

*****ERROR 060: Global operand already defined

Indicates that the global symbol was referenced before it was declared to be global. See GLOBAL directive explanation in Chapter 4.

*****ERROR 061: GLOBAL declaration Phase error

Indicates that a symbol was not declared in both passes of the assembler.

*****ERROR 062: Too many Sections and Globals

Indicates undefined globals, or more than 254 globals and sections defined.

*****ERROR 063: Invalid radix

Indicates an invalid radix character in the constant. The 9520 Software Development System recognizes only (H) hexadecimal, (O) or (Q) octal, and (B) binary radix codes.

*****ERROR 064: Invalid digit

Indicates an invalid digit in the indicated number base. For example, 10031B contains an invalid digit. Radix B indicates this to be a binary number, making digit 3 invalid.

*****ERROR 065: Unmatched string or parameter delimiter

Indicates an unmatched quotation mark delimiter or square bracket delimiter.

*****ERROR 066: Line too long after replacement

Indicates expanded line is too long. Only 128 characters are allowed.

*****ERROR 067: Extra replacement identifier

Indicates extra information following the replacement indicator in a macro definition block.

*****ERROR 068: Replacement invalid outside of Macro

Indicates improper use of replacement indicators #, @, and % outside of a macro definition block.

*****ERROR 069: Undefined replacement string

Indicates that the string variable has not yet been defined as a string.

*****ERROR 070: Invalid replacement identifier

Indicates that the replacement specification used is invalid.

*****ERROR 071: Scalar value required

Indicates an address value where a scalar value was required.

*****ERROR 072: Invalid expression

Indicates that the specified expression is either incomplete or inaccurate for the context required by the operation. Expressions are recognizable when the following values appear in the first item position of the operand: a variable, a constant, a left parentheses "(", a minus sign "-", or a logical not character "\".

*****ERROR 073: Section size Phase error

Indicates that the number of bytes generated for this section during the first pass is smaller than the number of bytes generated during the second pass.

*****ERROR 074: Undefined symbol

Indicates that a symbol in an expression has no value.

*****ERROR 075: String truncated

Indicates that the number of characters assigned to the string is greater than the string definition. See ASET Strings, Chapter 2.

*****ERROR 076: Negative SEG operand

Indicates a negative number in the operand of the SEG function. See SEG, Chapter 2.

*****ERROR 077: SEG starting operand is zero

Indicates a zero in the starting position of the SEG operand. See SEG, Chapter 2.

*****ERROR 078: Insufficient workspace

Indicates that a temporary data manipulation area has been exceeded. Could be caused by conditional assembly or string functions that leave too little memory to perform the required operations.

*****ERROR 079: Value too large

Indicates that the value of the space operand exceeds 255, and has been truncated.

*****ERROR 080: Invalid NAME symbol

Indicates that the symbol in the operand field of the NAME directive begins with a non-alphabetic character and is, therefore invalid.

*****ERROR 081: Illegally substituted ENDM

Indicates that an ENDM directive was substituted within the body of a macro expansion before the normal end of the macro is encountered.

*****ERROR 082: Nested INCLUDE directive

Indicates that the file inserted into the program with the INCLUDE directive contains another INCLUDE directive.

*****ERROR 083:Missing ENDIF

Indicates that a conditional IF block with a missing ENDIF directive was included in the program.

*****ERROR 084:Missing ENDM for included macro

Indicates that a macro definition block with a missing ENDM directive was included in the program.

*****ERROR 085: String value too large

Indicates that a string value to be used as a number exceeds two characters in length.

*****ERROR 086: Shift count exceeds 16

Indicates an attempt to shift right or left more than 16 bits.

*****ERROR 087: Too many symbols

Indicates a lack of room in the Assembler's symbol table to contain all symbols used by the program. The Assembler discontinued processing the program.

*****ERROR 088: Invalid transfer label

Indicates that a label used for the transfer address on an END directive is an unbound global, a scalar, or the result of a previous HI, LO, or ENDOF function.

*****ERROR 090: ENDOF applied to a bound GLOBAL

Indicates that the ENDOF function was used with a bound GLOBAL instead of a SECTION. In the case of an unbound GLOBAL, the function is resolved at link time.

*****ERROR 091: Unable to assign INCLUDE file

Indicates that MP/M or CP/M could not gain access to the file. This message will be accompanied by a message on the console during each pass. An SRB status code will indicate the reason that MP/M or CP/M could not assign the file.

The following error messages apply only to the Z80 Assembler:

*****ERROR 254: Invalid operand specification

The syntax of an operand is invalid, or the operand type is not valid for the current instruction, or the combination of operands is not valid for the instruction.

Error Messages applying only to the Z80 Assembler (Continued)

*****ERROR 253: Unmatched parentheses

A left parentheses which may specify 'contents of' does not have a corresponding right parentheses.

*****ERROR 252: Invalid index displacement

The displacement portion of an indexed operand is invalid.

*****ERROR 251: Too many elements in expression

In certain contexts parenthesized expressions or subexpressions may not contain more than 40 identifiers and, or string constants.

*****ERROR 250: Invalid operand combination

The combination of operands specified is not valid for the current instruction.

*****ERROR 249: Invalid branch condition for JR

A jump condition other than Z, NZ, C, or NC was specified for a JR instruction.

*****ERROR 248: Destination involves HI, LO, or ENDOF

The destination specified in a JR or DJNZ instruction involves the illegal use of one of the indicated functions.

*****ERROR 247: Relative jump out of current section

The destination of a JR or DJNZ instruction is not in the current section.

*****ERROR 246: Relative jump out of range

The destination of a JR or DJNZ instruction is not within the range -126 to +129 from the current instruction.

*****ERROR 245: Invalid bit position

The first operand of BIT, RES, or SET instruction did not specify a scalar value in the range 0-7.

*****ERROR 244: Invalid RST address

The operand of a RST instruction was either relocatable or was a scalar or absolute address whose value was not 0, 8, 10H, 18H, 20H, 28H, 30H, or 38H.

*****ERROR 243: IM operand is not scalar 0, 1 or 2

The operand of IM is invalid.

*****ERROR 242: Index displacement out of range

Index must be a byte value in range -128 to +127.

RESERVED WORDS

The Z80 Microprocessor instruction mnemonics, register symbols and MILLENNIUM SYSTEMS Assembler directive names must not be used as symbolic labels. The following names are reserved for these special uses:

Z80 INSTRUCTION MNEMONICS

ADC	CPD	DI	IN	JR	NOP	POP	RLA	RRCA	SRA
ADD	CPDR	DJNZ	INC	LD	OR	PUSH	RLC	RRD	SRL
AND	CPI	EI	IND	LDD	OTDR	RES	RLCA	RST	SUB
BIT	CPIR	EX	INDR	LDDR	OTIR	RET	RLD	SBC	XOR
CALL	CPL	EXX	INI	LDI	OUT	RETI	RR	SCF	
CCF	DAA	HALT	INIR	LDIR	OUTD	RETN	RRA	SET	
CP	DEC	IM	JP	NEG	OUTI	RL	RRC	SLA	

Z80 REGISTER SYMBOLS

A	H	BC	SP
B	I	DE	
C	L	HL	
D	R	IX	
E	AF	IY	

RESERVED FOR FUTURE USE

XREF

JUMP CONDITIONS

M (minus)	C (carry)
NC (noncarry)	PE (parity even)
NZ (non-zero)	PO (parity odd)
P (positive)	Z (zero)

MILLENNIUM SYSTEMS ASSEMBLER DIRECTIVES, OPTIONS & OPERATORS

ABSOLUTE	ELSE	IF	NCHR	SEG	WORD
ASCII	END	INCLUDE	NOLIST	SHL	
ASET	ENDIF	INPAGE	ORG	SHR	
BASE	ENDM	LIST	PAGE	SPACE	
BLOCK	ENDOF	LO	PAGED	STITLE	
BYTE	ENDR	MACRO	REPEAT	STRING	
CND	EQU	ME	RESERVE	SYM	
COMMON	EXITM	MEG	RESUME	TITLE	
CON	GLOBAL	MOD	SCALAR	TRM	
DEF	HI	NAME	SECTION	WARNING	