Tektronix®
COMMITTED TO EXCELLENCE

8002A
μPROCESSOR LAB
8080A/8085A
ASSEMBLER & EMULATOR USER'S MANUAL
Opt. 1,5,16,20,31&35

Tektronix, Inc.
P.O. Box 500
Beaverton, Oregon 97077

070-2702-01

Serial Number _____

First Printing JUL 1978

WARRANTY

The 8001/8002A μProcessor Lab System and options, excluding customer supplied equipment, is warranted against defective materials and workmanship, under normal use, for a period of ninety (90) days from date of shipment. CRTs found to be defective after the ninety (90) day period and up to twelve (12) months from date of shipment will be exchanged at no charge for the material. Tektronix will repair or replace, at its option, those System components which Tektronix determines to be defective within the warranty period.

In addition, in those areas where Tektronix has service centers available for this system, on-site warranty repair is provided at no charge during the first ninety (90) days from date of shipment.

Tektronix shall be under no obligation to furnish warranty service if:

a. Attempts to install, repair, or service the equipment are made by personnel other then Tektronix service representatives.

b. Modifications are made to the hardware or software by personnel other than Tektronix service representatives.

c. Damage results from connecting the 8001/8002A μProcessor Lab System to incompatible equipment.

There is no implied warranty for fitness of purpose. Tektronix is not liable for consequential damages.

CONTENTS

Page

SECTION 3 STATEMENT SYNTAX CONVENTIONS

SECTION 4 ASSEMBLER DIRECTIVES

Page

SECTION 4 ASSEMBLER DIRECTIVES (cont)

SECTION 5 MACROS

Page

SECTION 10 THE LINKER (cont)

SECTION 11 8080A/8085A SERVICE CALLS

SECTION 12 8080A/8085A DEBUGGING

SECTION 13 8080A REAL-TIME PROTOTYPE ANALYZER

SECTION 14 PROTOTYPE CONTROL PROBE

ABOUT THIS MANUAL

This manual explains 8002A μProcessor Lab assembly, linking, converting and emulation procedures for 8080A/8085A-based microcomputer development. The user should be familiar with hexadecimal and binary number systems, and with ASCII character code. It is especially helpful if your programming background includes assembly language experience.

The manual describes all Tektronix 8080A/8085A Assembler features and procedures in detail. These include: the basic source module format; all assembler directives; macro capability; assembled listing and object module formats; procedures for linking assembled object code; and procedures for converting Intel source code into Tektronix source code. Assembler operating procedures are discussed in this manual and in the source code. Assembler operating procedures are discussed in this manual and in the Tektronix 8002A μProcessor Lab System User's Manual.

The closing sections provide a detailed description of emulation procedures specific to 8080A/8085A-based microprocessor development, including 8080A/8085A service calls, debugging, and prototype control probe specifics.

The appendices contain essential summarized information and conversion tables. Appendix C is an alphabetical summary of 8080A/8085A assembly language instructions. Appendix F lists all error codes, messages, and their associated explanations.

Throughout this manual, zeros are slashed where needed for clarity.

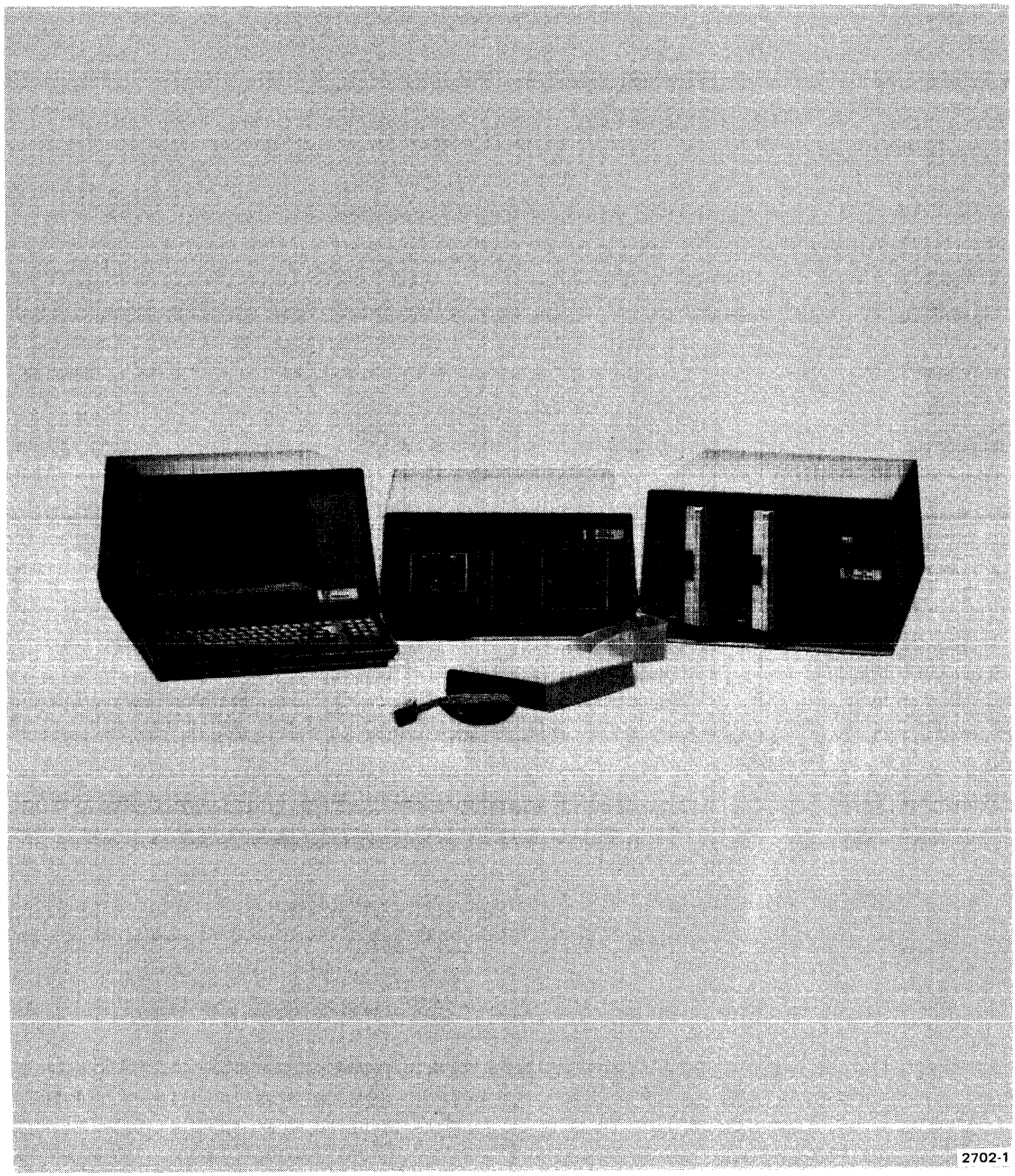


Fig. 1-1. The 8002A μProcessor Lab System with optional CT8100 CRT Terminal and 8080A/8085A Prototype Control Probe.

Section 1

TEKTRONIX 8080A/8085A

ASSEMBLER INTRODUCTION

ASSEMBLER INPUT

The Tektronix 8080A/8085A Assembler translates user-written programs (Assembler source files) into executable binary format (object files). The user's program must be written in 8080A/8085A symbolic notation (assembly language), and becomes the Assembler source file for Assembler operation. User-written programs can be entered into disc files with the text editor program, using procedures described in the Tektronix 8002A μProcessor Lab System User's Manual. If the Assembler source file is contained in more than one flexible disc file, each file name must be specified by assemble (ASM) command parameters.

All valid input devices can originate Assembler input. The Assembler reads the Assembler source file 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, Assembler listings, and appropriate information messages. The object module contains executable binary instructions and data constants translated from the source file. The entire object module must be linked, if it contains relocatable code, and then loaded into program memory in order to be executed on the 8080A/8085A Emulator Processor.

Assembler listings are composed of line numbers, the generated object code, and the source code as entered in the source file. Wherever an error is detected, an error code is printed on the display device and on 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 Assembler listing and object module to a disc, output file names are entered as ASM command parameters. To transfer Assembler listing and object modules to an output device (such as a line printer) instead of a file, the name of the device is specified as the ASM command parameter.

The Tektronix Assembler makes two passes through the source file. 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 file 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.

Section 2

ASSEMBLER SOURCE FILE FORMAT

INTRODUCTION

Symbolic 8080A/8085A instructions, Assembler directives, macro calls, and explanatory comments form the Assembler source file. Each 8080A/8085A source file statement must be entered according to the Tektronix 8080A/8085A Assembler format. When translated by the Assembler, the source file becomes the obejct module to be executed.

Three types of Assembler source file statements may be used:

1. 8080A/8085A symbolic instructions,
2. Assembler directives, and
3. macro calls.

8080A/8085A SYMBOLIC STATEMENT FORMAT

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

Each 8080A/8085A 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 8080A/8085A source file instruction is translated by the Assembler into one, two, or three, 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 file begins at a constant position within the line. This columnar format can be easily implemented by using the tab setting function to define field starting positions. Fig. 2-1 is an example of a properly formatted 8080A/8085A Assembler source file.

```
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      SET             4               ;DEFINE VARIABLE SYMBOL L2 TO EQUAL 4
        ORG             100H            ;STARTS OBJECT CODE OF NEXT
                                        ;INSTRUCTION AT 100H
        MOV             A,M             ;LOAD REG. A WITH CONTENTS OF MEMORY
                                        ;POINTED TO BY HL REGISTER PAIR.
        END                             ;END OF PROGRAM
```

Fig. 2-1. Properly Formatted 8080A/8085A Program.

A general description of the characteristics of each source file field follows. The entire 8080A/8085A instruction set is listed in Appendix C. The Tektronix Assembler directives are described in Section 4 and listed in Appendix B. Macro calls are described in Section 5.

The Label Field

Labels represent addresses associated with locations in a source file. Labels may be used in all 8080A/8085A instructions macro calls, and Assembler directives. Every label must be unique within each Assembler source file. Duplicate labels prevent proper program execution and cause an error code to appear on the display device and in the Assembler 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.

The EQU and SET directives are the only statements requiring label usage. In all other directives, label usage is optional. EQU and SET directives always equate the required label to the constant or expression value in the operand field. The SET 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 an 8080A/8085A 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 an 8080A/8085A 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 an 8080A/8085A instruction, an Assembler directive, or a macro call, the Assembler rejects the entire statement and prints an error code. Three 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 a 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 Sections 4 and 5, and summarized in Appendix B.

Seven types of information are permitted in the instruction operand field. Each instruction determines the operand types and their proper sequence. Refer to Appendix C for a summary of 8080A/8085A instruction requirements.

The following list defines the seven operand item types and their required syntax for 8080A/8085A instructions:

Operand Item Type	Operand Item Syntax
1) An 8080A/8085A 8-bit register containing the operand data.	B C D E H L A
2) An 8080A/8085A 16-bit register pair address containing the operand data.	B D H PSW SP
3) A memory indicator for the address containing the operand data and pointed to by the H,L register pair.	M
4) An 8-bit data or address constant within the range, −128 to 255. An immediate value.	Expression
5) A 16-bit data or address constant within the range, 0 to 65,535. An immediate value.	Expression
6) An 8-bit I/O device address within the range, 0 to 255.	Expression
7) A 16-bit absolute memory address within the range, 0 to 65,535, containing the operand data.	Expression

Several 8080A/8085A instructions operate on data in the seven 8-bit registers. The operand field for these instructions must contain the register symbol or register value for each register involved. Pre-defined register values are as follows:

Register Symbol	Register Value
B	0
C	1
D	2
E	3
H	4
L	5
A	7

The register pair symbols B, D, H, and PSW each represent two 8080A/8085A registers. The register pairs are used in combination and function as though they were one 16-bit register. Register pair symbols are as follows:

Register Symbol	Register Value	Register Content
B	ØH	High byte in B; low byte in C.
D	2	High byte in D; low byte in E.
H	4	High byte in H; low byte in L.
PSW	6	Register A is high byte. Low byte contains sign, zero, half carry, parity, and carry flags.

The SP register pair symbol represents the 16-bit stack pointer register. Like the PSW register pair, the SP register value is 6. Instructions using either the PSW or SP pointer register pairs are mutually exclusive so that no conflicts can arise.

The memory indicator symbol "M" can appear in any operand where a register symbol is acceptable. The M indicates that data is to be obtained from or stored to memory instead of a register. Memory location M is pointed to by the H,L register pair. For example, the instruction below stores the contents of register B to the memory location contained in registers H and L.

```
>MOV M,B
```

The symbol for the Assembler location counter contents "$" can appear in any operand item where a memory address is allowed. The $ represents the memory address of the first byte in the statement containing the $. For example, the following two instruction sequences are equivalent:

```
        LABEL   OPERATION       OPERAND         COMMENT

1)      TIMER   DCR             C               ;DECREMENT C REGISTER,
                                                ;LABEL INSTRUCTION TIMER
                JNZ             TIMER           ;JUMP BACK IF C NON-ZERO

2)              DCR             C               ;DECREMENT C REGISTER
                JNZ             $-1             ;JUMP BACK IF C NON-ZERO
```

The $ represents the address of the first byte in the JNZ instruction. Since the DCR 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 DCR and JNZ instructions without changing the $—1 expression. Inserting an instruction in the first example requires no other changes.

The symbols for the 8080A/8085A registers, register pairs, and memory address registers have been predefined by the Assembler. Any data constant, I/O device address, or memory 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 predefined by the Assembler (see Predefined Symbols in this section) must be defined in the label field of an EQU and SET directive or any 8080A/8085A instruction in the Assembler source file, 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 branches to itself. For example:

```
LABEL   OPERATION       OPERAND         COMMENT

HERE    JZ              HERE            ;HANG HERE IF LAST RESULT
                                        ;IS EQUAL TO 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 SET, 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 item requires an 8-bit value and the value represented is greater than 8-bits, 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 subsection entitled String Text Substitution and to Section 5 for discussion of 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 8080A/8085A instructions, macro calls, and Assembler directives, or in the operand field of GLOBAL, SECTION, COMMON, RESERVE, MACRO, or STRING directives. Once defined, symbols can be used in the operation and operand fields of 8080A/8085A instructions, macro calls, and Assembler directives.

A symbol label in an 8080A/8085A 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 predefined 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 Assembler source file. 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 SET symbol, however, may be redefined in another SET directive.

Pre-defined Symbols

Certain words are reserved as predefined 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.

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, and L.

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 PSW and SP. The register pair names are BC, DE, and HL.

3. The 8080A/8085A instruction mnemonics (refer to Appendices C and G).

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

5. The Tektronix Assembler directives reserved for future use (refer to Appendix G).

6. Memory indicator "M" referencing address pointed to by H,L register pair.

Refer to Appendix G for a complete list of reserved words for the 8080A/8085A Assembler.

Rules of 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 Ø through 9; and the special characters, . (period), — (underscore), and $ (dollar sign). Lowercase letters are interpreted in their uppercase form. A symbol may contain up to eight characters. Only the first eight characters of the symbol are used; excess characters are ignored. All predefined 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 (see Section 10). Upon assembly, address values are relative to an Assembler-defined section (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 Section 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 that 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 used:

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. For example:

   ØB5H and ØFFH

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 E 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 quote 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:

"ABC^DEF"	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 ("").

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 E.

String	Numeric Value
" "	Ø
"A"	41H
"12"	3132H
"123"	3132H (truncation error occurs

EXPRESSIONS PERMITTED IN THE OPERAND FIELD

The operand 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, a string 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 then described in greater detail, completing the 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
> =	greater than or equal
<	less than
< =	less 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 argument, subsequent operators must not be performed on the HI function result. The HI function result is numeric.

LO (exp)

Returns the lest 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 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. 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) \							
+	— (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 followig 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 following program segment demonstrates MOD operator usage.

```
LABEL   OPERATION       OPERAND         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
CX      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 following program segment 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
                                        ;ANY NUMBER SHIFTED 16 TIMES IN
                                        ;BINARY 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 in the following table.

	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 Ø to ØFFFFH (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 (ØFFFFH) 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 T
                                        ;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 E 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	Ø (false)
"A" > "B"	results in	Ø, since A is less than B
"ABC" > "AAAA"	results in	- 1, since B is greater than A
"ABC" > "ABC"	results in	Ø, since "ABC" has three characters and "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 not allowed. 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)+1);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 (ØFFFFH). 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 UNDEFINED
                                            ;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 1. 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. Its argument is the string expression whose length is to be determined. The result of NCHR is numeric and not a string value. The following examples illustrate NCHR function usage.

NCHR ("")	results in	Ø
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
        LXI             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 (ØFFFFH). Otherwise, the value of BASE is false (0). In the following examples, Q, R, and ZZ represent addresses where Q and R share a common base, while ZZ does not.

BASE (Q,R)	results in	ØFFFFH (true)
BASE (Q,Q+15)	results in	ØFFFFH (true)
BSSE (ZZ,Q)	results in	Ø (false)
BASE (Q,Q- R)	results in	Ø (false) because Q- R is scalar
BASE (5,15)	results in	ØFFFFH (true) because 5 and 15 are both scalar
BASE (5,Q- R)	results in	ØFFFFH (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 leftmost position in the resulting string. Examples of STRING function results follow:

STRING(5)	results in	"000005"
STRING(5+15)	results in	"000020"
STRING(ØFFH)	results in	"000255"
STRING(- ØFFH)	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 enchance 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 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 of the STRING directive, refer to Section 4, Assembler Directives.

SET Strings

The SET 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 SET 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 SET 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 LIIMITS OF
                                                ;2, 45, AND 0 CHARACTERS
A1      SET             "AB"                    ;VALUE OF A1 IS "AB"
A2      SET             A1                      ;VALUE OF A2 IS "AB"
A4      SET             A1:A2                   ;VALUE OF A4 IS ""
                                                ;TRUNCATION ERROR SINCE A4
                                                ;ALLOWS A VALUE LIMIT OF 0
                                                ;CHARACTERS
A3      SET             "A MEDIUM LONG STRING"  ;VALUE OF A3 IS "A
                                                ;MEDIUM LONG STRING"
A1      SET             A3                      ;VALUE OF A1 IS "A MEDIUM"
                                                ;STRING TRUNCATED
```

String Text Substitution

String variables may be used to modify source text being processed by the Assembler. Using string variables makes it possible to insert code into a source line; the code can 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      SET             "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 following line results upon assembly:

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

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

```
LABEL   OPERATION       OPERAND

        STRING          A1,A2,A3,A4
A1      SET             "YTE"
A2      SET             "123,456"
A3      SET             "COMMENT"
A4      SET             ""
```

The following substitutions are then performed.

```
SOURCE CODE             RESULTS AFTER SUBSTITUTION

BYTE'A1','A2'           BYTE YTE,123,456
WORD1'A4'               WORD 1
A4 SET"'A3'"            A4 SET "COMMENT"
WORD"'A4'"              WORD "COMMENT"
B'A1''A2'-200           BYTE123,456-200
B'A1"A2'                BYTE123,456 (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 to be performed. The caret character causes the single quote character to then be interpreted as a literal character in a statement. The following example demonstrates caret usage;

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

Section 3

STATEMENT SYNTAX CONVENTIONS

INTRODUCTION

Many of the following sections in this manual contain Tektronix Assembler and TEKDOS statement descriptions. Each statement description is preceded by a syntactical block showing the required statement format. This section describes the syntax conventions for Tektronix Assembler and TEKDOS statements.

TEKTRONIX ASSEMBLER STATEMENT SYNTAX

Tektronix 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 text. An item shown in lowercase 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 Section 2
2. expression—as defined in Section 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.

$$\left\{\{\text{strvar1}\}\ [\text{lenexp1}]\right\}$$

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 to the end of the line.

[,symbol] . . .

TEKDOS STATEMENT SYNTAX

A TEKDOS statement contains a command and in most cases, one or more parameters with delimiting characters. The following example shows a typical TEKDOS statement syntactical block.

Syntax

KEYWORD {filename}[device / filename [/disc drive]] [{line number 1} {line number 2}] . . .

Command Name

The TEKDOS command is the leftmost item in the syntax block.

A minimum set of characters (short form) is required for each TEKDOS command. This minimum character set is underlined in the syntactical description. In the page heading for the command, the exact spelling of the command name is given with the short form underlined. Commands without a short form are not underlined.

In addition to the minimum set of characters in the command name, a maximum set (long form) is also given for each command name. Any number of characters in the command name, ranging from the short form spelling to the long form spelling, may be used as long as the exact spelling is followed.

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 slash (/) is used to delimit a file name and the disc drive number.

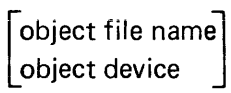
The comma may be used as a delimiter in most cases. Two commas are used to specify null or empty fields in a parameter list. Three commas are used to specify two adjacent null fields.

Parameters

The parameters or controlling conditions of each command line are shown in the TEKDOS statement syntactical block. These parameters may be names, numbers, characters, or symbols. When a parameter is shown capitalized, it must be entered exactly as shown. Parameters shown in lowercase letters are descriptive terms to signify the type of entry.

Braces and Brackets

Braces and brackets have the same meaning as when used with Tektronix 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.



Trailing Dots

As with Assembler directive syntax, a line of dots indicates the item can be repeated to the end of the line.

Section 4

ASSEMBLER DIRECTIVES

INTRODUCTION

The 8002A μProcessor Lab 8080A/8085A Assembler features the following directives.

Listing Format Control Directives

- LIST
- NOLIST
- PAGE
- SPACE
- TITLE
- STITLE
- WARNING

Symbol Definition Directives

- EQU
- STRING
- SET

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

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 a warning message.

Syntax

Label	*Operation*	*Operand*	*Comment*
[symbol]	LIST	[CND] [,TRM] [,SYM] [,CON] [,MEG] [,ME] [,DBG]	[;charstring]
[symbol]	NOLIST	[CND] [,TRM] [,SYM] [,CON] [,MEG] [,ME] [,DBG]	[;charstring]

Purpose

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

Explanation

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

Assembler Listing Format Control Options

Four general listing control options (CND, TRM, SYM, and CON) may be entered with the LIST directive, when specific features in the Assembler listing are desired for viewing. The same four listing options may be entered with the NOLIST directive, 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 that occur 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.

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 Section 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. Thus it is 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 unsatisfied 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 ME is OFF Status of both options is saved
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.

Linker Listing Format Control Option

DBG — Lists all global and local symbols in the Linker listing. Remains in the ON or OFF state until another LIST or NOLIST DBG is entered. See The Linker, Section 10.

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.

Example

The following listing control statement suppresses the symbol table listing in the Assembler 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 Asssembler listing.

```
LABEL   OPERATION       OPERAND         COMMENT

        LIST            ME,CND          ;LISTS MACRO EXPANSIONS AND ALL
                                        ;UNSATISFIED CONDITIONS
```

Syntax			
Label	*Operation*	*Operand*	*Comment*
[symbol]	PAGE		[;charstring]

Purpose

The PAGE directive causes the next Assembler 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 Assembler 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; 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.

Example

The following program illustrates PAGE directive usage:

```
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      SET             4               ;DEFINE VARIABLE SYMBOL L2
                                        ;TO EQUAL 4
        PAGE                            ;BEGINS NEW LISTING PAGE
        ORG             100H            ;STARTS OBJECT CODE OF NEXT
                                        ;INSTRUCTION AT 100H
        MOV             A,M             ;LOADS THE CONTENTS OF
                                        ;MEMORY INTO REG. A
        END                             ;END OF PROGRAM
```

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

```
TEKTRONIX 8080A/8085A ASM Vx.x                                    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      SET      4        ;DEFINE VARIABLE SYMBOL L2
                                           ;TO EQUAL 4

TEKTRONIX 8080A/8085A ASM Vx.x                                    PAGE 2

00005            0100 >  ORG      100H     ;STARTS OBJECT CODE OF NEXT
                                           ;INSTRUCTION AT 100H
00006    0100    7E      MOV      A,M      ;LOADS THE CONTENTS OF
                                           ;MEMORY INTO REG. A
00007                    END               ;END OF PROGRAM
.
.
.
TEKTRONIX 8080A/8085A ASM Vx.x    SYMBOL TABLE LISTING            PAGE 3

STRINGS AND MACROS

        S1-----0050 S

SCALARS

        A  --- 0007     B  --- 0000     C  --- 0001
        D  --- 0002     E  --- 0003     H  --- 0004
        L  --- 0005     L2 --- 0004V    M  --- 0006
        PSW--- 0006     SP --- 0006

% (default) SECTION 0001

        L1-----(0100)

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

Note that the symbol indicators V and S respectively follow the symbols L2 and S1. The symbol indicator V indicates that L2 is a SET 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 Section 7, entitled Assembler Listing Format.

Syntax

Label	*Operation*	*Operand*	*Comment*
[symbol]	SPACE	[expression]	[;charstring]

Purpose

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

Explanation

The expression in the SPACE directive operand field indicates the number of lines to be spaced downward. 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 Assembler 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.

Example

Assume the following source program resides on disc.

```
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      SET             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
        MOV             A,M             ;LOADS THE CONTENTS OF
                                        ;MEMORY INTO REG. A
        END                             ;END OF PROGRAM
```

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

```
TEKTRONIX 8080A/8085A ASM Vx.x                                    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      SET     4        ;DEFINE VARIABLE SYMBOL
                                          ;L2 TO EQUAL 4
                  .                .
                  .                .
                  .                .
                  .                .
                  .                .
                  .                .
                  .                .
                  .                .
                  .                .
00005       0100 >       ORG     100H     ;STARTS OBJECT CODE OF
                                          ;NEXT INSTRUCTION AT 100H
00006 0100 7E            MOV     A,M      ;LOADS THE CONTENTS OF
                                          ;MEMORY INTO REG. A
00007                    END              ;END OF PROGRAM
.
.
.
TEKTRONIX 8080A/8085A ASM Vx.x   SYMBOL TABLE LISTING             PAGE 2

STRINGS AND MACROS

        S1-----0050 S

SCALARS

        A  --- 0007     B  --- 0000     C  --- 0001
        D  --- 0002     E  --- 0003     H  --- 0004
        L  --- 0005     L2 --- 0004V    M  --- 0006
        PSW--- 0006     SD --- 0006

% (default) SECTION (0001)

        L1-----0100

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

Syntax			
Label	*Operation*	*Operand*	*Comment*
[symbol]	TITLE	{string expression}	[;charstring]

Purpose

The TITLE directive creates a text line at the top of each Assembler 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, including the carriage return. Any characters beyond 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:

```
TEKTRONIX 8080A/8085A ASM Vx.x   THIS IS THE PROGRAM TITLE      PAGE 1
```

Syntax			
Label	*Operation*	*Operand*	*Comment*
[symbol]	STITLE	{string expression}	[;charstring]

Purpose

The STITLE directive creates a text line on the second line of each Assembler 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 beyond the 72-character limit are truncated. The actual STITLE directive is not printed on the listing.

Example

Assume the following STITLE 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 as follows:

```
TEKTRONIX 8080A/8085A ASM Vx.x                                  PAGE 1

THIS LINE DEMONSTRATES STITLE USAGE
(blank line)
.
.  (source code)
.
.
```

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 001
```

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
```

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 SET NCHR ("ABD")
000D           WARNING    ****ENTRY OUT OF SEQUENCE
*****ERROR 001
```

SYMBOL DEFINITION DIRECTIVES

The Assembler symbol definition directives are presented in the order shown below:

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

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. The expression in the operand field represents the value of that name. The symbol acquires the same base as the operand expression. This symbol may not be re-defined.

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.
```

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 a numeric "lenexp" length expression is not specified, an eight-character maximum length is assumed. If "lenexp" expression is specified, it must be enclosed within parentheses. An operand symbol named in a statement that contains the optional character length expression must 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.

Example

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.
```

Syntax			
Label	*Operation*	*Operand*	*Comment*
{symbol}	SET	{expression}	[;charstring]

Purpose

The SET 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 a SET 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 SET 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 SET 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 a SET 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 the values are less than six characters long. The numeric-expression to string-symbol conversion process is diagrammed as follows:

```
LABEL    OPERATION        OPERAND          COMMENT

string   SET              numeric          ;RESULTS IN EXPRESSION
                                           ;CONVERSION TO STRING
```

If the SET directive contains a numeric-valued symbol and a string-valued expression, the string expression is converted to a numeric value. Refer to Section 2 of this manual, Assembler Source File Format, which describes string-to-numeric conversion. The string-expression to numeric-symbol conversion process is diagrammed as follows:

```
LABEL    OPERATION        OPERAND          COMMENT

numeric  SET              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.

Example

Examples of typical SET 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      SET             "AB"                    ;VALUE OF A1 IS "AB"
A2      SET             A1                      ;VALUE OF A2 IS "AB"
A4      SET             A1:A2                   ;VALUE OF A4 IS "",
                                                ;TRUNCATION ERROR SINCE
                                                ;A4 ALLOWS A VALUE OF
                                                ;ONLY 0 CHARACTERS
A3      SET             "A MEDIUM LONG STRING"  ;VALUE OF A3 IS "A MEDIUM
                                                ;LONG STRING"
A1      SET             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      SET             14              ;VALUE OF A1 IS "000014"
A2      SET             -1              ;VALUE OF A IS "-00001"
A1      SET             5EH             ;VALUE OF A1 IS "000094"
B1      SET             A2              ;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 SET VALUE OF
                                        ;2D30H
```

LOCATION COUNTER CONTROL DIRECTIVE

Syntax			
Label	*Operation*	*Operand*	*Comment*
[symbol]	ORG	{[/] 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

If an ORG directive is omitted at the beginning of a program, the Assembler location counter is set to Ø.

Omission of the optional / (slash) operator sets the location counter to the address specified by the operand expression. 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
```

Usage of the / operator in the operand field sets the location counter 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 "/Ø", the location counter is set to the next odd value. For example, when the current location counter contains 100H, and the following ORG directive is entered, the next instruction begins at location 101H.

```
ORG    /Ø
```

If the current location counter is already set to an odd value when the "/Ø" 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 /Ø is entered, and 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 Section 10, The Linker.)

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 Assembler 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      MOV             A,M             ;LOADS THE CONTENTS OF
                                        ;MEMORY INTO REG. A

.
.
.
```

Upon assembly, the Assembler listing lines for the preceding instructions appear as follows. The MOV instruction object code begins at location 100H. Notice the relocation indicator (>) on line 00005.

```
.
.
.
00005           0100 >      ORG     100H    ;STARTS OBJECT CODE OF NEXT
                                            ;INSTRUCTION AT 100H
00006   0100 7E  L1         MOV     A,M     ;LOADS THE CONTENTS OF
                                            ;MEMORY INTO REG. A
```

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.

Syntax			
Label	*Operation*	*Operand*	*Comment*
[symbol]	BYTE	{expression} [,expression]. . .	[;charstring]

Purpose

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

Explanation

Each data byte is represented by an expression. The data is stored in the Assembler 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, VALUE OF ASCII "2"
                                        ;TRUNCATION ERROR
```

Example

In this example, 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
```

Syntax			
Label	*Operation*	*Operand*	*Comment*
[symbol]	WORD	{expression} [,expression]. . .	[;charstring]

Purpose

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

Explanation

This directive is identical to the BYTE directive except that two bytes of program memory are allocated in the Assembler 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.

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
```

Syntax			
Label	*Operation*	*Operand*	*Comment*
[symbol]	ASCII	string expression [,string expression]. . .	[;charstring]

Purpose

The ASCII directive allows easy storage of text in program memory.

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.

Example

Assume the following lines of source code reside on disc:

```
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    SET         "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
```

The following hexadecimal object code is generated from the preceding source code.

```
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 E.

Syntax			
Label	*Operation*	*Operand*	*Comment*
[symbol]	BLOCK	{expression}	[;charstring]

Purpose

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

Explanation

The BLOCK operand expression indicates the number of bytes to reserve in program memory. The operand expression must be a positive value. Negative or invalid blocks change the location counter. 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 program memory storage block:

```
LABEL   OPERATION       OPERAND         COMMENT
        BLOCK           32              ;RESERVES 32 BYTES OF MEMORY
```

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 Section 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.

Syntax			
Label	*Operation*	*Operand*	*Comment*
[symbol]	MACRO	{symbol}	[;charstring]

Purpose

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

Explanation

A macro is a shorthand method for inserting a block of Assembler source code into a program one or more times. The MACRO directive names the Assembler 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 Assembler 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 Section 5, Macros.

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.

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 following the macro call in the source program.

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
        ENDM                            ; END OF MACRO DEFINITION
```

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" is not specified, the number of repeat cycles defaults to 255. Attempts to repeat beyond the value of "expression2" cause 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.

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   SET             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   SET             AGAIN+1         ;INCREMENT AGAIN AT ASSEMBLY TIME
        ENDR                            ;END OF REPEAT CONDITION
        BYTE            0DH             ;GENERATES CARRIAGE RETURN
        ENDM                            ;END OF MACRO DEFINITION
```

Syntax			
Label	*Operation*	*Operand*	*Comment*
[symbol]	INCLUDE	{string expression}	[;charstring]

Purpose

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

Explanation

When the INCLUDE directive is encountered, text from the file specified in the operand field is inserted into the Assembler source 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 Section 5, Macros, 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 Assembler source code, outside of macro definition blocks. When this occurs, the inserted text may contain macro definitions.

Example

The following example demonstrates INCLUDE directive usage.

```
LABEL   OPERATION       OPERAND         COMMENT

        .
        .
        .
        INCLUDE         "OTHFILE"               ;INSERTS OTHFILE INTO THE
                                                ;CURRENT PROGRAM AT THE
        .                                       ;ADDRESS OF THE CURRENT
        .                                       ;LOCATION COUNTER.
```

CONDITIONAL ASSEMBLY DIRECTIVES

The conditional assembly directives are presented in the order shown below.

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.

Syntax

Label	*Operation*	*Operand*	*Comment*
[symbol]	IF	{expression}	[;charstring]
[symbol]	ELSE		[;charstring]
[symbol]	ENDIF		[;charstring]

Purpose

The IF directive causes assembly of the Assembler 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.

Example

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
```

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 example demonstrates conditional macro termination with the EXITM directive.

```
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 SATISFIED
        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 Section 10, The Linker.

Mnemonic	Purpose
SECTION	Declares a program section, assigns a section name, and defines the section parameters.
COMMON	Declares a program 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. Its starting address is evenly divisible by 256. 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 there are no restrictions on where the Linker may place the section.

Syntax

Label	*Operation*	*Operand*	*Comment*
[symbol]	SECTION	{symbol} [,PAGE / ,INPAGE / ,ABSOLUTE]	[;charstring]

Purpose

The SECTION directive is used to declare an Assembler source 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 any given section is Ø. 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 Assembler 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 the previous discussion of Relocation Options in this subsection.) If no option is specified, the Linker considers the section to be byte relocatable.

When a label symbol is entered on the SECTION directive, the symbol represents address Ø, 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
```

Syntax

Label	*Operation*	*Operand*	*Comment*
[symbol]	COMMON	{symbol} [,PAGE / ,INPAGE / ,ABSOLUTE]	[;charstring]

Purpose

The COMMON directive declares an Assembler source program 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 Assembler source file. 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
```

Syntax

Label	*Operation*	*Operand*	*Comment*
[symbol]	RESERVE	{symbol, expression} [,PAGE / ,INPAGE]	[;charstring]

Purpose

The RESERVE directive is used to set aside a workspace in program 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 Assembler 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.

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
        .
        .
        .
```

Syntax			
Label	*Operation*	*Operand*	*Comment*
[symbol]	RESUME	[symbol]	[;charstring]

Purpose

The RESUME directive continues the definition of a given Assembler source file 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 Assembler object module name. When no object module 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
        .
        .
        .
```

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 Assembler source file may be referenced by another source module, after the modules are linked.

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.

If the modules are to be stored in the library file (see Section 9, The Library Generator) the value of each global symbol in the library file must be unique. 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.

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
        .
        .
        .
```

Syntax			
Label	*Operation*	*Operand*	*Comment*
[symbol]	NAME	{symbol}	[;charstring]

Purpose

The NAME directive declares the name of an Assembler object module.

Explanation

The symbol in the operand field of the NAME directive is the name assigned to the Assembler 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 from the NAME directive.

Example

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

```
LABEL   OPERATION       OPERAND         COMMENT

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

MODULE TERMINATION DIRECTIVE

Syntax			
Label	*Operation*	*Operand*	*Comment*
[symbol]	END	[expression]	[;charstring]

Purpose

The END directive terminates Assembler source files.

Explanation

The END directive terminates a source file contained in one or more disc files. A source file 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 TEKDOS 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.

Section 5

MACROS

INTRODUCTION

A macro is a shorthand approach for inserting Assembler source code into a program. A macro is often used when the same, or nearly the same block of code is repeatedly used within a program. A block of macro code is called a macro definition block. The source code that results from a macro definition block may be altered each time the macro is called. Thus, 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 longer than, the macro call.

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 examined. Then, each phase of the process is presented in greater detail.

BASIC MACRO EXPANSION PROCESS

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

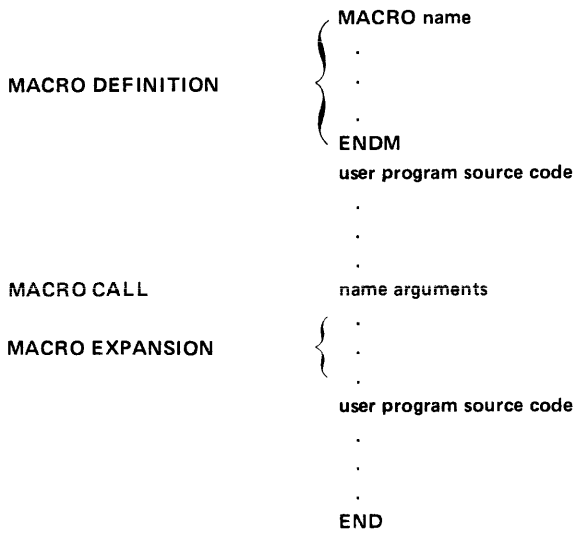


Fig. 5-1. The macro expansion process.

As mentioned, there are three phases of macro usage; definition, calling, and expansion. First the macro must be defined. The macro is named in a macro definition directive. The directive is followed by 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 an Assembler source 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 must be alphabetic. The macro name must be unique from all symbols in an Assembler source program.

MACRO DEFINITION BLOCK

The lines following the macro definition directive, up to and including an ENDM directive, become a predefined block of code. The block of code is 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 replaement 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 ('1') upon macro expansion.

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

The parameter within single quotes ('N') may be either a number or a numeric-valued SET directive. This capability is discussed in Section 4, Assembler Directives, describing the SET 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 may be used only 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 '@' 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.

Reserved words, however, should not be used with the '@' character.

The # Character

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       SET             1               ;INITIALIZES J TO EQUAL 1
                                        ;AT ASSEMBLY TIME
        REPEAT          J<='#'          ;REPEAT WHILE J IS LESS THAN
                                        ;OR EQUAL TO 3
J       SET             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.

In the following example, 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  SET             "'%'"           ;SECNAM IS SET TO NAME OF
                                        ;CURRENT SECTION
        SECTION         BBB             ;DEFINES NEW SECTION BBB
        .
        .
        .
        RESUME          'SECNAM'        ;RESUMES PREVIOUS SECTION
        .
        .
        .
```

The ↑ or ∧ Character in Macro Definition

The up-arrow (↑), or caret (∧), character may be entered just prior to any character having special meaning, thus causing the special character to be interpreted as a regular part of the text. The ↑ or ∧ is available in all phases of the Tektronix Assembler. In the following example, 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. The operand field of a macro call contains the macro name to be called.

```
LABEL   OPERATION       OPERAND

        name
```

INCLUDE Directive Text Insertion

Another method for calling text into a program involves INCLUDE directive usage. The INCLUDE directive (see Section 4, Assembler Directives) may be used to insert text into a program from a specified file. The INCLUDE directive may be 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         filename
```

Text Substitution

Optional arguments within the operand field of the macro call are separated by commas. These arguments 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 [ ] 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. Square brackets are not passed to the source text during macro expansion. For example, the following macro call parameters

```
LABEL   OPERATION       OPERAND                 COMMENT

        PNPDG           ABC,1"ABC,1",[ABC,1]    ;INVOKES MACRO PNPDG AND
                                                ;SUBSTITUTES THE ARGUMENTS
                                                ;ABC,1,"ABC,1",ABC,1
```

produce the following arguments.

Argument 1 = ABC

Argument 2 = 1

Argument 3 = "ABC,1"

Argument 4 = ABC,1

The ↑ or ∧ Symbol in Macro Calls

The up-arrow (↑), or caret (∧), character may be entered just prior to any character having special meaning, thus causing that character to be interpreted as a regular part of the text. The ↑ or ∧ symbol is available in all phases of the Tektronix Assembler. 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 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. However, 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.

```
LABEL   OPERATION       OPERAND

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

The preceding example expands to the arguments listed below. 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 the line does not exceed 128 characters (not including a carriage return). When 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, one of which 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
```

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   SET             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'
```

This macro call generates the following 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, which calls the macro Q1 and appears as Q1 7. This statement causes expansion to continue with the statement, PARM1 SET 1, which allows PARM1 to be used as a symbolic reference to the first parameter. The next statement within Q1 is expanded as BYTE 3,5,7, replacing BYTE 3,5,'PARM1'. Expansion within macro Q1 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 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 Tektronix 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 below. For further information pertaining to IF—ELSE—ENDIF and REPEAT—ENDR usage, refer to Section 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.
2)	REPEAT	expr1,expr2	If expr1 is equal to zero (false), statements up to the ENDR statements 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 another IF—ENDIF or REPEAT—ENDR block.

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 PARAMTER
        ENDM                             ;END OF MACRO DEFINITION
```

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 terminated further macro expansion.

REPEAT—ENDR Blocks

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

```
LABEL   OPERATION       OPERAND         COMMENT

        MACRO           CONDR           ;DEFINES CONDR AS MACRO NAME
AGAIN   SET             1               ;INITIALIZES 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   SET             AGAIN+1         ;INCREMENT AGAIN AT ASSEMBLY TIME
        ENDR                            ;END OF REPEAT CONDITION
        BYTE            0DH             ;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
```

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 example, there are three arguments specified and the '#' character is replaced by 3.

```
                0001    AGAIN   SET     1
                FFFF    +       REPEAT  AGAIN<=00003
          0000  19      +       BYTE    25
                0002    AGAIN   SET     AGAIN+1
                                ENDR
                FFFF    +       REPEAT  AGAIN<=00003
          0001  1A      +       BYTE    26
                0003    AGAIN   SET     AGAIN+1
                                ENDR
                FFFF    +       REPEAT  AGAIN<=00003
          0002  1B      +       BYTE    27
                0004    AGAIN   SET     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. However, blank lines or comment lines 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 then inserted and assembled within the main program.

Before it assembles each line in the macro definition block, the Assembler scans for the presence of single quotes. An argument defined in the macro call then replaces the parameter within the single quotes. 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.

Section 6

ASSEMBLER OPERATING PROCEDURES

INTRODUCTION

This section describes the syntax required for the Tektronix Assembler to translate source code into executable binary object code.

Syntax

```
        [object filename [/disc drive]]  [list filename [/disc drive]]
ASM     [object device                ]  [list device                ]

        {source filename [/disc drive]}  [source filename [/disc drive]]
        {source device                }  [source device                ] ...
```

PURPOSE

The ASM command invokes the Assembler when the 8002A μProcessor Lab is under TEKDOS control.

EXPLANATION

The optional filename or device parameter causes the Assembler to output the binary object module to the specified disc file or device. The optional list filename or device parameter causes the Assembler to output an Assembler listing to the specified device or disc file. The source filename or device parameter specifies the source file to be translated.

All parameters within the ASM command line must be separated either by spaces or by commas. The object filename or device parameter is optional and, if omitted, must be replaced by two commas in the following manner. In this case an object file is not generated.

```
ASM,,LIST SOURCE
```

The list filename or device parameter is also optional and, if omitted, must be replaced by two commas in the following manner. In this case an Assembler listing is not generated.

```
ASM OBJECT,,SOURCE
```

If the object and list filenames or devices are both omitted, they must be replaced by three commas in the following manner.

```
ASM,,,SOURCE
```

If the object and list files are intended to reside on a disc other than the system disc, the appropriate disc drive number must follow the slash character (/) in the following manner.

```
ASM OBJECT/1 LIST/1 SOURCE
```

At least one source filename or device must be specified in the ASM command line. More than one source filename or device may be specified if the ASM command and its parameters do not exceed one line. If the source file is stored on a disc other than the system disc, the appropriate disc drive number must be specified after the / character in the following manner.

```
ASM OBJECT LIST SOURCE/1
```

If the specified source file is a device, the Assembler source code must be entered twice, once for each assembler pass. In addition, if the source file is the console input device (CONI), care should be taken to ensure that the source code is entered exactly the same for both Assembler passes.

ASSEMBLY COMPLETION

After assembly completion, each line containing an error is displayed, followed by an error code describing the nature of the error. Refer to Appendix F for a list of error codes, messages, and their explanation. Below all error displays, two lines appear on the output device showing the number of source lines, the number of assembled lines, the number of available bytes, and the number of errors and undefined symbols. If an irrecoverable assembly error occurs, the program aborts and a message indicates the error in the following form:

```
FATAL ERROR, ASSEMBLY ABORTED AT LINE XXXX
```

The TEKDOS prompt character (>) appears after all Assembler messages have been displayed indicating assembly completion.

If an object filename or device parameter has been specified in the ASM command line, the translated program is stored as relocatable binary object code. A correctly assembled object file may be linked, if necessary, and then loaded, executed or debugged.

If a list filename or device parameter has been specified in the ASM command line, the assembled listing is output to a device or disc file.

Section 7

ASSEMBLER LISTING FORMAT

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 information, and error responses related 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:

```
TEKTRONIX 8080A/8085A ASM Vx.x                                    PAGE X
```

With the TITLE directive, 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, as follows. Refer to the TITLE directive in Section 4, Assembler Directives.

```
TEKTRONIX 8080A/8085A ASM Vx.x    THIS IS THE PROGRAM TITLE       PAGE X
```

With the STITLE directive, 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. Refer to the STITLE directive in Section 4, Assembler Directives.

```
TEKTRONIX 8080A/8085A 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, depending 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 code 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 if present or the object code 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 source code follows the relocation or substitution indicator (if present) or the object code field. 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 character, 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 in the following format.

```
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 F.

If the error response precedes an additional message, "FATAL ERROR, ASSEMBLY ABORTED AT LINE XXXX", the error is so severe that the Assembler cannot continue execution.

THE SYMBOL TABLE

The symbol table follows the listing, and indicates 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:

```
TEKTRONIX 8080A/8085A 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. All register values are also described here.
name SECTION characteristic (length)	All symbols based to the name 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 Section 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.
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 vaules 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 is 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:

```
SYM1 ---0101   SYMB2 --0025   SYMB --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 SET 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	SET 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.

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 Fig. 7-1.

```
TEKTRONIX 8080A/885A ASM Vx.x                                    PAGE 1

                                STRING  S1(80)   ;DEFINE STRING VARIABLE S1
                                                 ;WITH 80-CHARACTER MAXIMUM
              0003      L1      EQU     3        ;DEFINE CONSTANT SYMBOL L1
                                                 ;TO EQUAL 4
***** ERROR     003     Symbol value Phase Error
              0004      L2      SET     4        ;DEFINE VARIABLE SYMBOL L2
                                                 ;TO EQUAL 4
              0100      >       ORG     100H     ;STARTS OBJECT CODE OF NEXT
                                                 ;INSTRUCTION AT 100H
      0100    7E00      L1      MOV     A,M      ;LOAD REG.A WITH CONTENTS
                                                 ;OF MEMORY. MULTIPLY-DEFINED
                                                 ;SYMBOL, L1.
***** ERROR     002     Symbol already defined
                        END                      ;END OF PROGRAM

.
.
TEKTRONIX 8080A/8085A ASM Vx.x    ASSEMBLER SYMBOL TABLE          PAGE 2

STRINGS AND MACROS

        S1-----0050 S

SCALARS

        A  --- 0007     B  --- 0000     C  --- 0001
        D  --- 0002     E  --- 0003     H  --- 0004
        L  --- 0005     L2 --- 0004V    M  --- 0006
        PSW--- 0006     SP --- 0006

% (default) SECTION (0101)

        L1-----0100
 13 SOURCE LINES      15 ASSEMBLED LINES      23 BYTES AVAILABLE
  2 ERRORS
```

Fig. 7-1. Sample Assembler and symbol table listing.

Section 8

ASSEMBLER OBJECT MODULE APPLICATION

INTRODUCTION

The Tektronix Assembler object module can be stored on flexible disc in binary code. The binary object code may then be linked or loaded into program memory for execution and debugging. If a module contains more than one section or references GLOBAL symbols declared in other modules, it must be linked before its object code is loaded into program memory.

PROGRAM LOADING AND EXECUTION

The TEKDOS command, LOAD, is used to load an assembled binary object module or linked load file into program memory. The TEKDOS command, GO, may then be entered to begin program execution or debugging. This section outlines LOAD and GO command usage. For further details describing binary object code execution procedures, refer to the 8002A μProcessor Lab System User's Manual.

Syntax

LOAD [/offset] {filename [/disc drive]} filename [/disc drive] . . .

PURPOSE

The LOAD command program loads Assembler object modules and Linker load files into program memory.

EXPLANATION

The specified files are loaded into program memory with the LOAD command. The files must have been previously created by the Assembler or the Linker. Assembler object files containing relocatable sections or references to global symbols may execute incorrectly if not linked before loading.

The specified files are loaded into program memory starting at the location specified in the source code.

The offset amount alters the LOAD memory address.

Syntax

GO [address]

PURPOSE

The GO command causes execution control to be passed to the emulator processor.

EXPLANATION

This command passes execution control to the emulator processor. The first time you enter the GO command without an address parameter, execution begins at address 0. Thereafter, if you enter the GO command without an address parameter, execution begins at the address following the last instruction executed. When you enter the GO command with an address parameter, execution begins at the specified address in program memory.

Execution is suspended by pressing the ESC key. Breakpoints set with the BKPT command interrupt program execution when the program reads from or writes to the specified address.

When a transfer address is specified with the Assembler directive END, the LOAD command passes this transfer address to the GO command. Execution will begin at the transfer address when the GO command is specified without an address parameter. The GO command may be invoked in or out of the debug system and still have control passed to the transfer address.

Section 9

THE LIBRARY GENERATOR

INTRODUCTION

The Library Generator, LIBGEN, is a general purpose library-generating routine for the 8002A μProcessor Lab.

LIBGEN collects TEKDOS Assembler object modules into library files. These modules can then be retrieved by the Linker as needed. Each object module must be entered into a library file from a separate object file.

LIBGEN also allows the modification of an existing LIBGEN generated library file. Modules may be inserted into, deleted from, replaced, and extracted from disc files. More than one library file may exist at a time.

INVOKING LIBGEN

LIBGEN can be invoked in two different ways: command file or interactive.

Command File Invocation

Syntax

LIBGEN @file

LIBGEN will read the commands from "@file" until either an end-of-file is encountered or the END command is read.

If any errors are detected in the command file, LIBGEN will abort with the message: "ERRORS IN COMMAND FILE, LIBGEN ABORTED."

Read the following subsection, LIBGEN, for a list of allowable commands for your command file.

Interactive Invocation

Syntax

LIBGEN [newlib] {,} [listfile [/disc drive] / listdevice] {,} [oldlib]

"newlib" is the name of a new library file to be generated. To omit, replace with a comma. For example:

```
LIBGEN      ,,listfile,oldlib
```

"listfile" is the name of the listing file. The listing file contains a summary of action taken, any error messages, and a summary of the contents of the new library file. If omitted, no listing file will be generated. To omit, replace with a comma. For example:

```
LIBGEN      newlib,,oldlib
```

To omit both "newlib" and "listfile", enter:

```
LIBGEN,,,oldlib
```

"listdevice" is any output device.

"oldlib" is the name of an old library file to be modified. This parameter must be entered if an old library file exists.

When the LIBGEN command line is entered, an asterisk (*) prompt character will appear, indicating that LIBGEN is ready to accept commands. Enter the desired LIBGEN commands. See the following subsection, LIBGEN Commands, for a list of allowable commands. To process the library file and return to TEKDOS, enter the END command.

LIBGEN COMMANDS

The TEKDOS syntax rules apply to LIBGEN commands. Refer back to Statement Syntax Conventions, in the Preface. Commands may be entered in either the short or the long form. For example, INSERT may be entered *I or *INSERT. Do not attempt to enter commands in any other form. For example, *INSER and *INS are not acceptable forms for the INSERT command.

Each command must be terminated by a carriage return.

NEWLIB {filename}

"filename" is the name of a new library file to be generated. Only one new "filename" is allowed. If this command is entered more than once, the name assigned to the file will be the one in the last command.

LIST {filename [/disc drive] / device}

"filename" or "device" will contain a summary of action taken, any error messages generated, and a summary of the new library file. Only one list file is allowed. If this command is entered more than once, the name assigned to the file will be the one in the last command. See the LIBGEN OUTPUT discussion later in this section for listing format.

OLDLIB {filename[/disc drive]}

"filename" is the old library file to be modified. Only one old library file may be modified per LIBGEN invocation. If this command is entered more than once, the name assigned to the file will be the one in the last command.

INSERT {file1[/disc drive]} [,file2[/disc drive]] ... [BEFORE libmodule / AFTER]

The object modules in "file1" ",file2"... will be inserted into the new library file.

If the "BEFORE/AFTER libmodule" parameter is not entered, the object files will be inserted at the beginning of the library file.

If the "BEFORE/AFTER libmodule" parameter is entered, the object files will be inserted before or after the module named "libmodule" in the library file.

If the "BEFORE/AFTER libmodule" parameters are entered, but the module named libmodule cannot be found in the old library file, the object files are inserted at the end of the new library file. A warning message is given. See the Errors subsections.

If the BEFORE/AFTER parameter is entered without specifying a libmodule, a syntax error will occur.

If duplicate file names are entered, a warning message will be printed on the console, and modules will be inserted as requested. See the Errors subsection.

If different modules contain symbols with identical names, errors may occur at link time.

DELETE {mod1} [,mod2] [,mod3]

The DELETE command will prevent modules "mod1", "mod2", "mod3"... from being copied from the old library file into the new library file.

If duplicate module names are present in the library file, the DELETE command will delete all modules with that name.

REPLACE {libmodule BY file [/disc drive]}

The REPLACE command is a combination of the DELETE and INSERT BEFORE commands. The library file is scanned for the module "libmodule". The location of the module after "libmodule" is noted, "libmodule" is deleted, and the object module in "file" is inserted before the module that followed "libmodule".

If duplicate module names exist in the library file, the REPLACE command will delete all modules with that name. It will then place the new file where the first module was.

Using the REPLACE command with a non-existing "file" deletes the "libmodule".

If different modules contain global symbols with identical names, the wrong module may be linked at link time.

EXTRACT {libmodule TO file [/disc drive]}

The library module "libmodule" will be extracted from the old library file and placed in an object disc file named "file". The library module will still exist in the old library file. If a file named "file" already exists on the disc, it will be replaced by the new file.

NOLOG

LIBGEN will not print the LIBGEN commands to the listing file or the console (in the case of command file invocation). LIBGEN defaults to LOG (see the following discussion).

LOG

LIBGEN will print the LIBGEN commands to the listing file, or the system console. LIBGEN defaults to LOG.

END

LIBGEN will begin processing all the LIBGEN commands.

LIBGEN EXECUTION

LIBGEN will read all commands and collect information into its own internal data structures until the END command is read. It will then process the commands in the following order:

INSERT BEFORE

EXTRACT

DELETE

INSERT AFTER

LIBGEN makes two passes over each object module. During the first pass, the Library Module Block will be written and the Section Definition, Entry Point Definition, and the Global Definition records from the Global Symbol Directory block will be combined and written as a Symbols Defined Here (SDH) Block.

The symbols in the SDH Block identify the module for the Linker. Make sure that each global symbol in each module of the library file is unique.

On the second pass, the entire Assembler object module will be copied into the new library file.

Modules must be placed in the library file in the correct order. A module that references global symbols must precede the module that defines them. Otherwise, errors may occur at link time.

If more than one module with the same name is entered into the library file, a warning message will be sent to the listing file and the console. The Linker will link the first module by that name in the library file, and will ignore all subsequent modules with the same name.

LIBGEN OUTPUT

The optional listing is a three-part summary. Part one shows the LIBGEN commands used in the last execution. Part two contains a list of all modules now in the library file, in order, and their symbols. Section symbols are given an "S" prefix, entry point definitions, an "E"; and global symbols, a "G". Part three is a summary of modules extracted, deleted, and inserted in the last execution. Refer to the examples at the end of this section.

ERRORS

Non-Fatal Errors

Entering faulty input commands, or specifying non-existent files or modules, will cause an error or warning message to be displayed. Processing will continue

WARNING. DUPLICATE MODULE NAME: mod1

The module "mod1" exists more than once in the library.

MODULE(S) NOT FOUND IN oldlib

The modules listed after this message were not found in the old library file "oldlib".

UNABLE TO ASSIGN file1

The object file "file1" is not on the disc. Check the spelling of the file name and the disc number.

COULD NOT FIND MODULE mod1 IN oldlib, newmod INSERTED AT END OF newlib

The module "mod1" was not found in the old library file "oldlib." The module "newmod" has been inserted at the end of the new library file "newlib." See the INSERT and REPLACE command decriptions.

NO OLD LIBRARY GIVEN, file INSERTED AT END OF newlib

The name of the old library file was not entered. The module in "file" has been inserted at the end of the new library file, "newlib". See the REPLACE and INSERT command descriptions.

SYNTAX ERROR

The command is not of the correct form.

ILLEGAL COMMAND

The command is illegal. Check the spelling.

INVALID FILE NAME

The file name contains an invalid character.

INDIRECT FILE DEPTH EXCEEDED

A command file cannot contain the name of another command file.

Fatal Errors

Improperly formatted object files, an improperly formatted old library file, lack of symbol blocks in a module, and other system problems will cause an error message to be displayed. Processing will stop.

command DATA STRUCTURE OVERFLOW

"command" may be INSERT, DELETE, or EXTRACT. Too many modules are specified in the input commands. A maximum of 100 modules may be INSERTed, DELETEd, or EXTRACTed, or a maximum of 150 for any combined operation.

oldlib NOT A LIBRARY

"oldlib" is not a LIBGEN-generated library file.

CAN NOT FIND END BLOCK FOR MODULE IN FILE file1

Module in "file1" is not of the proper format.

CAN NOT FIND END BLOCK FOR MODULE mod1 OF LIBRARY oldlib

Module "mod1" of the old library file "oldlib" is not of the proper format. "mod1" cannot be copied into the new library file from "oldlib."

FILE file1 IS NOT AN OBJECT FILE

Object format for file "file1" is incorrect.

filename I/O ERROR #nn

"nn" is the TEKDOS SVC status byte.

INVALID OBJECT FORMAT FOR FILE file1

LOCATION = nnnn

The file "file1" located at address "nnnn" is not an object file.

RESTRICTIONS

A library file may contain a maximum of 100 modules. During one execution of LIBGEN, a maximum of 100 modules may be INSERTed, DELETEd, or EXTRACTed. A combination of up to 150 INSERTs, DELETEs, and EXTRACTs may be requested.

LIBGEN will accept only one command line at a time. Continuation lines are not allowed. Lines may be a maximum of 128 characters from the console or file.

To execute LIBGEN, an 8002A μProcessor Lab with 32K of program memory is required. To selectively link modules from the library file, version 2.0, or a later version of the LINKER is required.

EXAMPLES

Example 1

Create a new library file where no old library file exists. Output the listing to the line printer:

```
>LIBGEN NEW,LPT1

*I ABIO/1 MPIO/1

*INSERT STSO/O

*END
```

The listing file for this execution will be printed on the line printer (LPT1), as follows:

Tektronix Library Generator Vx.x COMMAND LOG Page 1

I ABIO/1 MPIO/1

INSERT STSO/O

END

Tektronix Library Generator Vx.x SYMBOLS DEFINED Page 2

MODULE: (*modulename1)

(S) (names of any sections) (G) (names of any global symbols) (E) (names of any entry point symbols)

MODULE: (*modulename2)

(S) (names of any sections) (G) (names of any global symbols) (E) (names of any entry point symbols)

MODULE: (*modulename3)

(S) (names of any sections) (G) (names of any global symbols) (E) (names of any entry point symbols)

Tektronix Library Generator Vx.x SUMMARY OF ACTION Page 3

NEW LIBRARY GENERATED: NEWL1

MODULE:	"modulename1"	FROM	ABIO/1	INSERTED
MODULE:	"modulename2"	FROM	MPIO/1	INSERTED
MODULE:	"modulename3"	FROM	STSO/0	INSERTED

*See NAME in the Assembler Directives section of this manual.

Example 2

Modify the library file "NEWL" using a command file:

```
>LIBGEN @FIXLIB
```

(where the file "FIXLIB" contains the following commands:)

```
LIST LIBLST

NEWLIB NEW1

OLDLIB NEWL

INSERT SPSO/1

END
```

The console will display the following message:

```
NEWLIB NEWL1

OLDLIB NEWL

INSERT SPSO/1

END
```

The listing file, "LIBLST" will contain the following:

Tektronix Library Generator Vx.x COMMAND LOG Page 1

NEWLIB NEWL1

OLDLIB NEWL

INSERT SPSO/1

END

Tektronix Library Generator Vx.x SYMBOLS DEFINED Page 2

MODULE: (modulename4)

(S) (names of any sections) (G) (names of any global symbols (E) (names of any entry point symbols)

MODULE: (*modulename1)

(S) (names of any sections) (G) (names of any global symbols) (E) (names of any entry point symbols)

MODULE: (*modulename2)

(S) (names of any sections) (G) (names of global symbols) (E) (names of any entry point symbols)

MODULE: (modulename3)

(S) (names of any sections) (G) (names of any global symbols) (E) (names of any entry point symbols)

Tektronix Library Generator Vx.x SUMMARY OF ACTION Page 3

NEW LIBRARY GENERATED: NEWL1

MODULE: "modulename4" FROM SPSO/1 INSERTED

*See NAME in the Assembler Directive section of this manual.

Section 10

THE LINKER

INTRODUCTION

The Linker merges independently assembled sections into an 8002A file, suitable for loading into memory. Linker input may come from the Assembler, and from the library files. (See Section 9, The Library Generator, for information on the library files.)

The 8002A μProcessor Lab Assembler converts user-written source files into machine language object modules. Each module consists of one or more sections.

The object modules output from the Assembler consist of Text Blocks, Relocation Blocks, and Global Symbol Directory Blocks. Text Blocks from an independently assembled program section consist of three types of information.

1. Constants and machine instructions whose values are independent of their position in memory;

2. Addresses or address constants whose values are relative to the starting location (base) of a section; and

3. Global references to other object modules whose values cannot be determined until all sections are assigned memory locations.

Text blocks are in binary data form.

Relocation Blocks contain information necessary to update and relocate bytes of program text. Global Symbol Directory Blocks define global symbols and sections.

The Linker supports the unique qualities of each of the microprocessors supported by the 8002A. The Linker's outward appearance and its operational method remain the same, regardless which microprocessor is supported.

To prepare object modules for the 8002A LOAD program, the Linker performs three specific functions for each module, in the order of entry:

1. allocates memory space for each section of the load file;

2. establishes a reference table of global symbols; and

3. when necessary, relocates address-dependent locations to correspond to allocated space.

In addition, the Linker generates a listing that indicates where sections are allocated, and states the values of all global symbols.

LINKER INVOCATION

Three methods of Linker invocation are available: simple invocation, interactive command invocation, and command file invocation. Simple invocation requires entry of filenames only; all other parameters are set to reasonable default values. This method is usually adequate for most linking situations.

For more precise control, the Linker can be invoked by an interactive command series using default or user-specified parameters. The user can specify section attributes and section location, define global symbols, and control the listing content.

In command file invocation, the Linker is activated by specifying a file containing a Linker command series.

Simple Invocation

Syntax

```
LINK [load file[/disc drive]] [list file [/disc drive]  ] {object1 [/disc drive]     }
                              [list device              ]  {LIB(libfile [/disc drive])}

                                  [,LIB(libfile [/disc drive]]
                                  [,object1 [/disc drive]    ] ...
```

The "load file" represents the name to be assigned to the Linker-created load module. The "list file" represents the listing file name. "list device" is the name of any output device used to print out the listing. "object1" represents any object file output by the Assembler. "LIB(libfile)" must be entered if any of the object code to be linked is in the library file (see The Library Generator). "libfile" is the name of the desired library file. The parentheses around "libfile" are required. Object and library files must be entered in the command line in the proper order, or linking errors may occur.

With simple invocation, all filename parameters must be entered on one line. No other parameter entries are permitted. If filenames for "load file", "list file", or "list device" are not entered, the corresponding file is not generated. If the list file or load file is not entered, it must be replaced by commas, as follows.

LINK,, list file object file

(Load file omitted, replaced by two commas.)

LINK load file ,, object file

(List file omitted, replaced by two commas.)

LINK ,,, object file

(Load and list file omitted, replaced by three commas.)

A map and error messages are output to the list file, and error messages are also logged to the console.

Command File Invocation

Syntax

LINK {**@filename**}

Commands are read from "filename" until an end-of-file or an END command is encountered. End of file or END directs the Linker to discontinue command mode and begin processing object modules. If errors have been generated, the Linker aborts with the message: ERRORS IN INDIRECT FILE, LINK ABORTED.

See the Linker Commands subsection for a list of legal commands for your command file.

LINKER COMMANDS

The following commands may be used in interactive or command file invocation.

LOG

Print messages to the console and log commands on the list file, if one has been specified. All commands are echoed to the Linker list file after LOG has been indicated.

NOLOG

Do not log Linker messages on the console.

MAP

Generate a memory map in the Linker list file. A memory map lists module names, section names and attributes, entry points within sections, and undefined global symbols. (See the Linker Output description in this section.)

NOMAP

Do not generate a memory map

LIST

Generates a Linker list file named "filename". See the listing file description for contents of the Linker list file. "filename" is any valid TEKDOS file specification. Instead of a filename, any valid output device may be designated.

LOAD {filename}

Generates a load file named "filename". The file will contain the executable output of the Linker and can be loaded using the LOAD command.

DEFINE {symbol1 = value} [,symbol2 = value] . . .

Define symbols. "symbol1", "symbol2". . . are names of global symbols. "value" is a hexadecimal number.

LINK $\left\{\begin{matrix}\text{object1 [/disc drive]} \\ \text{LIB(libfile [/disc drive])}\end{matrix}\right\}$ $\left[\begin{matrix}\text{,LIB(libfile [/disc drive])} \\ \text{,object1 [/disc drive]}\end{matrix}\right]$ . . .

Link modules. This command directs the Linker to include the specified object modules in the load file. The object modules and library modules must be entered in the correct order.

LOCATE {section name} [,BASE (starting address) / ,RANGE (starting address, ending address)] [,PAGE / ,INPAGE / ,BYTE]

Locate a section and/or redefine its relocation type. Note that redefining the relocation type of a section may cause the linked code to execute differently than intended.

section name

The name of the section to be allocated.

BASE

The hexadecimal starting address.

RANGE

The hexadecimal starting and ending addresses. If there is not enough space within the specified range, the section will not be linked.

PAGE

A relocation type; causing relocation at the beginning of a physical block of memory.

INPAGE

A relocation type causing relocation on any byte address, provided the section does not extend across page boundaries.

BYTE

A relocation type causing relocation at any byte address.

Relocation type is originally specified during assembly. (See SECTION, COMMON, and RESERVE Assembler directives; and Relocation Options.) If not specified before assembly, the section is byte-relocatable. If ABSOLUTE was specified before assembly, the relocation type may not be re-defined during linking, and the section may not be re-located.

@filename

Indicate indirect command file. This command directs the Linker to obtain subsequent commands from "filename." Commands are read from "filename" until an end of file or an END command is encountered. Indirect commands are echoed on the console as they are read, if LOG is specified. Nested indirect command files are illegal; a command file may not contain an "@filename" command.

TRANSFER {symbol / value}

Specify load module transfer address. "symbol" is a global symbol and "value" is a hexadecimal number with a leading character ranging from 0 through 9. This transfer value supersedes any transfer address encountered in linking object modules.

END

End command entry mode. If no errors have been generated in command file invocation, this command will terminate command entry mode and initiate the processing of object modules. If errors are detected, an appropriate message is issued and control is returned to the system console.

Command Processing Errors

EXTRANEOUS INFORMATION IGNORED

Extra characters are on a command line that only requires an instruction (e.g., LOG, NOLOG, MAP). The Linker performs the appropriate action for the command, ignoring extra characters on the line.

ILLEGAL COMMAND

The command was not recognized.

SYNTAX ERROR

Statement syntax is invalid. This error occurs when a command is incorrectly formed. For example, unmatched parentheses are found in the LOCATE command, or an operand is missing after the equals sign in the DEFINE command.

INDIRECT FILE DEPTH EXCEEDED

A filename command was found during processing of an indirect command file. The command is ignored.

INVALID FILE NAME

The file in a LIST, LOAD, or LINK command contains illegal file characters, the filename may not begin with a numeric character (0—9). One to eight characters from the following set are acceptable:

Alphabetic (A—Z), numeric (0—9), or special characters ("# & ' ( ) * ; = ?). An optional two-character disc drive indicator (/0 or/1) can follow the filename.

NOTE

Processing of the command line ceases when an invalid filename is encountered. All files up to the invalid filename, in the case of the LINK command, are added to the list of files to be linked.

INVALID RANGE SPECIFIED

The range (starting address through ending address) in the LOCATE command is invalid. The ending address must be greater than the starting address.

Examples

If an error is detected during command entry, a caret (^) is printed below the line to indicate the error location. A message defining the error is also printed. The following are examples of errors during interactive command entry. Linker-generated characters are underlined.

```
*LINK FILE1 FILE2 3FILE
           ^
INVALID FILE NAME

*LIST LISTFILE
*DEFINE A==BB
          ^
SYNTAX ERROR

*DEFIN SYMBOL=3
 ^
ILLEGAL COMMAND

*LOG NO PARAMETERS NEEDED
     ^
EXTRANEOUS INFORMATION IGNORED
```

If these errors have been contained in a command file, and if the LOG command had been activated, the errors would have been logged to the Linker listing.

LINKER EXECUTION

Program Sections

A section is a collection of object code that has been assembled with the same location counter. An object module may consist of several sections. These sections are treated separately by the Linker and each section is independently relocatable. No limit is placed on the number of sections per link, but no more than 255 sections or globals may exist in any one object module.

A section has five attributes that provide the Linker with information regarding memory allocation and where to link the section. These attributes are name, section type, size, relocation type, and memory location.

NAME — A section has a name consisting of up to eight characters, assigned by the section directives, SECTION, RESERVE, or COMMON, at assembly time. The name must be a valid identifier. The section name is entered into the Linker's symbol table and is a valid external symbol.

SECTION TYPE — A section may be either a SECTION, RESERVE, or COMMON. The specification is made through use of the SECTION, RESERVE, or COMMON directive at assembly time.

Each SECTION name must be unique. Multiple SECTIONs with the same name will be flagged as errors, and only the first one will be linked.

RESERVE sections with the same name are concatenated by the Linker. The length of a RESERVE section in a load module is the sum of all RESERVE sections with the same name.

COMMON sections with the same name are allocated the same space in memory. The length of the linked COMMON is that of the largest COMMON section.

SIZE

The size of each section in an object file is determined at assembly time. Section size is the number of program memory bytes that the section may occupy.

RELOCATION TYPE

A section may be absolute (non-relocatable) byte relocatable, page boundary relocatable, or inpage relocatable.

An absolute section is not relocated by the Linker. Memory locations in an absolute section where code has been generated, or where locations have been explicitly reserved by the Assembler BLOCK directive, are not allocated to any relocatable section at link time. However, if two or more absolute sections have code at the same address, the contents of those memory locations after linking are undefined. These memory conflicts, if they occur, are noted on the Linker memory map.

A byte relocatable section can be placed anywhere in memory.

The two remaining relocation types are allocated according to page boundaries. A page is a physical block of memory having a size and a starting address. The size of a memory page is microprocessor-dependent. 8080A/8085A pages are 256 bytes long. All pages start on boundaries evenly divisible by the page length. Page size 256, for example, implies pages starting at 0, 256, 512, ...

A page boundary relocatable section is allocated memory, starting on a page boundary.

An inpage relocatable section may be linked on any byte boundary, as long as it does not span two or more pages. If an inpage relocatable section is longer than the microprocessor page length, the Linker generates an error, redefines the program section to be page-boundary-relocatable, and continues linking.

MEMORY LOCATION

At link time the user may specify a relocatable section location, in the form of either a base address or an address range where the section may be placed. The default range for a relocatable section is the entire address space of the microprocessor. If the user elects not to specify a location for a section, the Linker will locate the section. An absolute section cannot be moved at link time.

The Default Section

If no SECTION directive is entered before assembly, the entire module is considered to be a byte-relocatable section with the same name as the object module.

Memory Allocation of Sections

The linker allocates memory in the following sequence:

1. Absolute sections.
2. Based sections.

   Based means a program section starting location has been specified by a LOCATE command.

3. *Ranged page relocatable sections

   Ranged means the user has explicitly declared a RANGE (starting address, ending address) with the LOCATE command at link time.

4. *Ranged inpage-relocatable sections.
5. *Ranged byte-relocatable sections.
6. Page-boundary-relocatable sections.
7. Inpage-relocatable sections.
8. Byte-relocatable sections.

Absolute and based sections are linked even if conflicts occur. A conflict exists when two or more sections have bytes at the same address. Other section types are not linked if a conflict occurs. If any memory conflict occurs during allocation, the conflict is noted on the memory map. The content of memory in the conflicting area is undefined.

Endrel

ENDREL is a pre-defined symbol whose value is assigned at link time. After memory is allocated, ENDREL is assigned the value of the first memory address available for use. This address is one greater than the highest address used by a non-based relocatable section. All relocatable sections are located below the value of ENDREL. Absolute sections, or sections relocated using the LOCATE command with a BASE specified, may or may not be located above the ENDREL address.

The user can override the value of ENDREL by assigning any other value to ENDREL. If ENDREL is neither defined nor referenced, no value is assigned.

*Range was declared at link time.

Linking the Library File

A portion of the object code for linking may reside in the library file (see The Library Generator). The Linker scans the Symbols Defined Here (SDH) blocks of each module in the library file. If it finds a symbol that matches one in the Linker symbol reference table, it links the module for that SDH block. It is important to define all global symbols uniquely if they are to be stored in the library file. Otherwise, the wrong module may be linked. It is also important to link the files and library modules in the correct order. (See The Library Generator.)

LINKER OUTPUT

Linker Listing File

The listing file may be output either to a flexible disc file or to the console, line printer, or other output device.

The following information may be included in a Linker output listing:

	Command Invocation	Simple Linker Invocation
Global Symbol List	Yes	Yes
Internal Symbol List	If specified	If specified
Map	If specified	Yes
Linker Statistics	Yes	Yes
Error Messages	If specified	Yes

Global Symbol List

A global symbol list is an alphabetical list of all global symbols (sections and symbols) and their assigned values. If a symbol is undefined, its value field contains asterisks.

```
TEKTRONIX 8080A/8085A LINKER Vx.x   GLOBAL SYMBOL LIST                    PAGE X

ABSECT2 0000    DO_IO   3700    ENTRY1   4091    ENTRY2   43A1
ENTRY3  0090    ENTRY4  0450    INPUT    3A00    MAINPROG 3E41
NOTHERE ****    OUTPUT  3B50    RELSECT2 0400    RELSECT3 2500
STACK   3600
```

In the preceding example, the global symbol NOTHERE was undefined, but was referenced by one or more input object modules.

Internal Symbol List

The internal symbol list contains all symbols in the source file and their actual values. The list consists of three parts:

1. Scalars.
2. Alphabetical list of labels for each section.
3. Alphabetical list of labels for each unbound global.

If there are no labels for a section or global, then no list for that section or global is output.

The internal symbol list will be displayed only if the BDG Parameter was entered with the LIST directive before assembly.

Example

```
TEKTRONIX 8080A/8085A LINKER Vx.x  INTERNAL SYMBOL LIST                PAGE X

FILE:    OBJECT FILE

MODULE:  MODULE NAME

  SCALARS:

        SYMBOL1    NNNN    SYMBOL2    NNNN    SYMBOL3    NNNN    SYMBOL4    NNNN
            .
            .
            .
        SYMBOLN    NNNN

  LABELS:   (SECTION   SECTION NAME 1)

        LABEL1     NNNN    LABEL2     NNNN    LABEL3     NNNN    LABEL4     NNNN
            .
            .
            .
        LABELN     NNNN

  LABELS:   (SECTION   SECTION NAME N)

        LABEL1     NNNN    LABEL2     NNNN    LABEL3     NNNN    LABEL4     NNNN
            .
            .
            .
        LABELN     NNNN

  LABELS:   (GLOBAL   GLOBAL 1)

        SYMBOL1    NNNN    SYMBOL2    NNNN    SYMBOL3    NNNN    SYMBOL4    NNNN
            .
            .
            .
        SYMBOLN    NNNN

LABELS:   (GLOBAL   GLOBAL N)

        SYMBOL1    NNNN    SYMBOL2    NNNN    SYMBOL3    NNNN    SYMBOL4    NNNN
            .
            .
            .
        SYMBOLN    NNNN
```

Map

A map consists of two parts: a module map and a memory map.

A module map is a listing of modules linked into the load file. The map contains information concerning sections and global symbols defined in each module.

```
TEKTRONIX 8080A/8085A Vx.x    MODULE MAP

FILE:  FILE

MODULE:  MAINMOD
         DO-IO            SECTION BYTE 3700-3E40
         INPUT            3A00  OUTPUT    3B50
         MAINPROG         SECTION BYTE 3E41-5141
         ENTRY1           4091  ENTRY2    43A1
         STACK            RESERVE PAGE 3600-36FF


FILE:  FILE2

MODULE:  OBJFILE2
         ABSECT2          SECTION ABSOLUTE 0040-0357
         ENTRY3           0090
         RELSECT2         SECTION PAGE 0400-2400
         ENTRY4           0450

FILE:  FILE3

MODULE:  OBJFILE3
         RELSECT3         SECTION PAGE 2500-3500
```

The module map lists linked modules. An alphabetical list of sections and entry points appears for each module. If no sections were linked in a module, an appropraite message so indicates. If no room for section is available, a *NO ROOM* message will be displayed. If a section was empty, an *EMPTY* message will be displayed.

A memory map is an ordered listing of the memory allocated to sections. The list starts with the lowest allocated address and proceeds to the highest allocated address space of linked sections.

```
TEKTRONIX 8080A/8085A LINKER Vx.x     MEMORY MAP

0040-0357        ABSECT2          SECTION ABSOLUTE
0400-2400        RELSECT2         SECTION PAGE
2500-3500        RELSECT3         SECTION PAGE
3600-36FF        STACK            RESERVE PAGE
3700-3E40        DO_IO            SECTION BYTE
3E4A-5141        MAINPROG         SECTION BYTE
```

Addresses are starred '*' if a conflict (an overlap) with another section occurred during allocation. Section type is either SECTION, COMMON, or RESERVE. Relocation type is either PAGE, INPAGE, BYTE, or ABSOLUTE.

Linker Statistics

The Linker Statistics include the number of errors, the number of undefined symbols, the number of sections, the number of modules, and the transfer address.

```
1 ERROR          1 UNDEFINED SYMBOL
3 MODULE         6 SECTIONS

TRANSFER ADDRESS IS GO40
```

The TRANSFER ADDRESS identifies program starting location. After loading the program in this example, the appropriate command would be "GO 40".

Error Messages

Three classes of errors can be generated during Linker execution:

WARNINGS (W)

A problem may exist but the linked program can probably be executed.

ERRORS (E)

Linked program probably will not execute properly.

FATAL ERRORS (F)

Errors directly affecting the Linker's execution. The Linker closes all open channels and returns control to TEKDOS.

All errors cause a message to be output to the LOG and LIST file or device. A fatal error will be output to the console even if NOLOG was specified.

In the following list, each error message is indicated as being a Warning (W), an Error (E), or a Fatal Error (F).

F. LINKER INTERNAL ERROR AT nnnn

An error occurred in the Linker. Try linking again. If this error persists, carefully document the incident and submit an LDP Software Performance Report to Tekronix.

E. NO ROOM IN RANGE nnnn-nnnn FOR SECTION name

The section length is greater than available contiguous memory in range nnnn-nnnn of allocated section memory.

W. SECTION name CHANGED FROM INPAGE TO(BYTE/PAGE)RELOCATABLE

Section length is greater than the page size of the microprocessor. This could occur if several inpage reserve sections were linked together and their total size exceeded the page size of the microprocessor. A section declared to be inpage relocatable, in a LOCATE command, will generate this error if the section exceeds microprocessor page size. If section size exceeds available page size, relocation will then be to a byte boundary.

F. INVALID OBJECT CODE FORMAT FOR FILE name LOCATION = nnnn

The information in file is not valid input object format. Ascertain that all files to be linked have been assembled. Location is the internal Linker address where the object file error was detected.

F. UNABLE TO ASSIGN file or device name

A file name specified as an input object module does not exist, or file device is unavailable.

F. MEMORY FULL

Linker memory is totally allocated and linking has been terminated. The total number of globals, sections, or object modules must be reduced in order to link in the available memory.

W. TRANSFER ADDRESS UNDEFINED

No transfer address was specified to the Linker either through the TRANSFER command or by specifying "END (expression)" during assembly. When no transfer address is specified, the Linker creates transfer address 0.

W. TRANSFER ADDRESS MULTIPLY DEFINED IN MODULE name FILE name

The module has attempted to redefine the transfer address previously specified by a linked module or by the TRANSFER command. The Linker uses the first encountered transfer address to generate a transfer address for the load module. If no transfer address is specified, a transfer address of 0 is generated.

W. RELOCATION TYPE OF SECTION name MULTIPLY DEFINED IN MODULE name FILE name

An attempt was made to redefine the section relocation type (byte, page, inpage, or absolute). This occurs when the LOCATE command defined a relocation type differing from that specified at assembly time. The error also occurs when relocation attributes of a COMMON or RESERVE section differ between modules. The Linker uses the first encountered relocation attribute to define the section.

E. Symbol name MULTIPLY DEFINED IN MODULE name FILE name

Indicates that an attempt was made to redefine a global symbol or section. This error occurs when two modules both define a global of the same name or when two sections have the same name. Code section names must be unique. In the event of multiply-defined sections, the Linker will only include the first one in the load module.

W. TRUNCATION ERROR AT nnnn IN MODULE name FILE name

The relocated value computed for LO byte relocation is too large to fit into one byte.

E. UNRESOLVED REFERENCE AT nnnn MODULE name FILE name

A reference to an undefined global or section was specified at this point in the object code. This occurs when a global is used in one module but was never defined. The unresolved reference is filled with zeros in the load file.

W. MACHINE REDEFINED FROM microprocessor IN MODULE name FILE name

The current input module has been generated for a different microprocessor than the previous object modules. Differences between microprocessor definitions may cause incompatabilities during linking (e.g., page length, alignment, etc.).

E. SECTION name EXCEEDS MAXIMUM SIZE

Section length is greater than the address space of the microprocessor. The section is not included in the load file. This error may occur when a Reserve is too long. The maximum size for the 8080A/8085A is 64K bytes.

W. IMPLICIT REORIGIN TO 0 IN SECTION name IN MODULE name FILE name

The Linker processed an object file where code in an absolute Section wrapped around from location FFFFH to 0.

W. SECTION name CHANGED FROM PAGE RELOCATABLE

Either:

1. the section was declared to be page relocatable and the Linker does not support paging for that microprocessor; or

2. there was insufficient room for a paged section in available memory. The Linker will attempt to allocate memory for the Section on a Byte Relocatable Boundary.

F{LIST FILE / LOAD FILE / CONSOLE / COMMAND FILE / OBJECT FILE} I/O ERROR #nn

This error indicates that the Linker was unable to read to or write from the specified file or device. The error number corresponds to the SVC status byte.

W. ATTEMPT TO RE-DEFINE FILE TYPE FOR filename

"filename" was specified twice: once as an object file and once as a library file. The Linker uses the first file type specified.

The Load File

The primary output from Linker processing is the Load file. A Load file is a subset of the Linker input object modules with all references and relocation resolved. It consists of a Module Block, a Global Symbol Directory Block, Relocation Reserve, and Text Blocks, and symbol table blocks, (if present in input modules) followed by an END Block. Load files are read into program memory with the LOAD command.

Section 11

8080A/8085A SERVICE CALLS

INTRODUCTION

A service call (SVC) allows the 8080A/8085A Emulator Processor to obtain peripheral service from the system processor during program execution. The SVC is an instruction sequence in the user program containing:

1. an 8080A/8085A output instruction, including the address of the emulator processor output port; and

2. a no-operation instruction, allowing time for the SVC to occur.

This section gives SVC information specific to the 8080A/8085A Emulator Processor. A broader description of SVC's is given in the 8002A μProcessor Lab System User's manual.

The SVC references the emulator processor output port address and cues the system processor that an I/O (input/output) function is to occur. The system processor then references a service request block pointer in the user program. The service request block (SRB) pointer in turn references a block of memory containing the actual service request I/O specifications. The I/O specification block is called the service request block (SRB). The SRB contains parameters such as:

1. the type of I/O to be performed,

2. the I/O device or file channel assignments, and

3. the size of buffers for data trasfer.

With these parameters, the service call can be executed within a defined SVC buffer area.

SVC procedures specific to the 8080A/8085A Emulator Processor are described in this section. The specific procedures describe the way the 8080A/8085A SVC output instruction refers to the SRB pointer address pair. The table below shows the SRB pointer address pair referred to by each 8080A/8085A SVC output instruction.

SVC	8080A/8085A SVC Output Instruction and Address	SRB Pointer Address Pair	
1	0F7H	0040H	0041H
2	0F6H	0042H	0043H
3	0F5H	0044H	0045H
4	0F4H	0046H	0047H
5	0F3H	0048H	0049H
6	0F2H	004AH	004BH

The 8080A/8085A SVC Operation

The 8080A/8085A SVC operation is initiated with the 8080A/8085A instructions "IN" and "OUT". The "IN" and "OUT" instructions reference the SRB pointer, which in turn references the appropriate SRB. The SRB then defines the peripheral I/O operations, and the buffer area where the I/O is to be performed. In the final step the peripheral I/O is performed within the defined buffer area.

An example of the 8080A/8085A SVC process follows. The program, named NEWPROG, uses an SVC that causes an ASCII line to be read into the SRB I/O buffer from the console input device. After the line is read into the buffer, the program halts. The comments to the right of each instruction explain the SVC execution sequence.

```
;PROGRAM TO READ ASCII DATA FROM THE CONSOLE INPUT DEVICE AND HALT
        SECTION         EXAMPLE,ABSOLUTE        ;DECLARES SECTION NAMED
                                                ;EXAMPLE TO BE RELOCATABLE
CONFOR  ORG             0                       ;BEGINNING ADDRESS OF SVC
                                                ;LABELED CONFOR
;THE NEXT TWO LINES COPMPOSE THE SVC
        OUT             0F7H                    ;SVC1
        NOP                                     ;ALLOWS TIME FOR SVC TO OCCUR
        HLT                                     ;PROGRAM HALTS AFTER SVC IS
                                                ;COMPLETE
        ORG             040H                    ;BEGINNING ADDRESS OF SRB
                                                ;POINTER
;THE NEXT TWO LINES COMPOSE THE SRB POINTER
        BYTE            HI(CONSRB)              ;RESULT IS HI BYTE FOR SVC1
        BYTE            LO(CONSRB)              ;RESULT IS LO BYTE FOR SVC1
        ORG             100H                    ;BEGINNING ADDRESS OF SRB
;THE NEXT EIGHT LINES COMPOSE THE SRB
CONSRB  BYTE            1H                      ;READ ASCII AND WAIT
        BYTE            1H                      ;CHANNEL NUMBER 1
        BYTE            00                      ;STATUS
        BYTE            00                      ;SINGLE BYTE DATA
        BYTE            00                      ;BYTE COUNT
        BYTE            CONIRD+1                ;BUFFER LENGTH
        BYTE            HI(CONBUF)              ;HI BYTE OF BUFFER POINTER
        BYTE            LO(CONBUF)              ;LO BYTE OF BUFFER POINTER
        ORG             200H                    ;BEGINNING ADDRESS OF BUFFER
CONIRD  EQU             80                      ;MAX INPUT LINE LENGTH LESS CR
;THE FOLLOWING LINE DEFINES THE SRB BUFFER AREA
CONBUF  BLOCK           CONIRD+1;               ;DEFINES BUFFER FOR SVC
        END             CONFOR                  ;SPECIFIES STARTING
                                                ;INSTRUCTION IN PROGRAM
```

To assemble and load the program, enter the following command lines:

```
>ASM NEWOBJ NEWLIST NEWPROG

>LOAD NEWOBJ
```

Assign channel 1 to the console input device.

```
>ASSIGN 1 CONI
```

This assignment corresponds to the channel byte assignments in the preceding SRB.

Now execute the program.

```
>GO 0
```

The desired character string "STRING" is entered and read from the console input device as follows:

```
STRING
```

The ASCII characters S, T, R, I, N, and G are then stored in the buffer.

The DUMP command may be used to display the hexadecimal contents of the buffer. The beginning address of the buffer was defined in the program as 2ØØH.

```
>DUMP 200

0200=53  54  52  49  4E  47   0D  XX  XX  XX  XX  XX  XX  XX  XX  XX
      |   |   |   |   |   |
      S   T   R   I   N   G   (carriage return, followed by previous
                               contents of program memory)
```

Section 12

8080A/8085A DEBUGGING

INTRODUCTION

Five debugging commands support the unique 8080A/8085A Emulator Processor architecture, and thus require special mention. These commands are summarized below, in the order in which they are presented in this section. For further debugging information, refer to the Debug Command in the 8002A μProcessor Lab System User's Manual.

The 8080A contains only one accessible interrupt function. The function can be enabled and disabled by using the EI and DI instructions. (See Appendix C, Interrupt and Control Instructions.)

The 8085A adds three more programmable interrupts, plus interrupt masking capabilities. A non-programmable TRAP function is also added.

These three new interrupts may be masked by using the SIM instruction. They may be enabled by the use of IE.

NOTE

A clock failure will cause DEBUG to be aborted. The system will remain in the same emulation mode.

Command Name	8080A Debugging Command Summary
TRACE	Enables or disables program execution monitoring. When TRACE is enabled, program execution trace lines display the current instruction location, its hexadecimal representation, mnemonic, and operands. Trace lines also show the contents of the stack pointer and certain other register values.
DSTAT	Display line shows the current status of the debugging session. The display line shows the emulator processor's next instruction address, all active breakpoints and their parameters, the current emulation mode, and the contents of the two-byte stack pointer and all other registers.
SET	Reassigns hexadecimal values to the two-byte stack pointer (SP), and registers labeled RF, RA, RB, RC, RD, RE, RH, and RL.
DISM	Disassembles object code from program memory into assembler mnemonics and hexadecimal operand.

Syntax

TRACE

or

TRACE { ALL / JMP / OFF } [STEP] [[lower address] [upper address]]

PURPOSE

The TRACE command enables or disables program execution monitoring.

EXPLANATION

When TRACE is enabled, program execution trace lines display the location of the current instruction, its hexadecimal representation, mnenonic, and operands. Trace lines also show the contents of the stack pointer and register values, labeled in the following order.

SP, RF, RA, RB, RC, RD, RE, RH, and RL.

The Trace Modes

TRACE displays all TRACE commands currently in effect. When TRACE ALL or TRACE JMP is entered in the DEBUG mode, displayed trace lines allow program execution flow monitoring. TRACE ALL causes trace information for all instructions executed by the emulator processor to be displayed on the DEBUG display device.

TRACE JMP causes trace information to be displayed each time program execution flow is altered. Conditional branches are traced only if the conditions are met.

If the STEP option is entered with either the TRACE ALL or TRACE JMP command, and a program is executed, control is returned to the DEBUG display device, allowing programmer intervention after each instruction's trace line is displayed.

When TRACE OFF is entered, all trace display is disabled

The "lower address" parameter marks the beginning of the block to be treated. The default is 0. The "upper address" parameter marks the end of the block. It must be equal to or greater than the "lower address" parameter. The default value is FFFF.

The Trace Line

Each trace line resulting from TRACE ALL or TRACE JMP contains one program instruction and information pertinent to its execution. Displayed trace lines appear in the following format:

LOC INST MNEM OPER SP RF RA RB RC RD RE RH RL

All trace line values are displayed in hexadecimal format. Definitions of the elements of the 8080A trace line follow:

LOC The location of the last executed instruction

INST The hexadecimal representation of the last executed instruction

MNEM The instruction mnemonic

OPER The operands

SP The stack pointer value

RF The flag register value

RA The value of register A

RB The value of register B

RC The value of register C

RD The value of register D

RE The value of register E

RH The value of register H

RL The value of register L

Trace Line Termination

In TRACE ALL or TRACE JMP mode, trace lines of all instructions or all branch instructions, respectively, are continuously displayed during program execution. Tracing stops when one of the following occurs: (1) an end of job condition is reached, (2) a breakpoint suspends the display, (3) the space bar is pressed to suspend the display, (4) the HLT instruction suspends the display, or (5) the ESC key is pressed to suspend program execution.

The ESC key may be pressed while the display has been suspended by a HLT instruction, in emulation modes 1 and 2. To reenter the TRACE mode, enter the following command.

GO [address]

Execution then continues at the beginning of the HLT instruction, if a GO "address" is not specified.

EXAMPLE

Suppose the following 8080 assembly language user program resides on your work disc:

```
LABEL   OPERATION       OPERAND         COMMENT

START   ORG             00
        XRA             A               ;CLEAR ACC
        MOV             B,A             ;CLEAR REG. B
        MOV             H,A             ;CLEAR REG. H
        LXI             D,13FFH         ;LOAD TOP OF MEMORY
        LDAX            D               ;LOAD 1ST NUMBER
        DCX             D               ;POINT TO NEXT NUMBER
        MOV             C,A             ;PUT 1ST NUMBER IN REG. C
        LDAX            D               ;LOAD 2ND NUMBER
        DCX             D               ;DECREMENT POINTER
        MOV             L,A             ;PUT 2ND NUMBER IN REG. L
        DAD             B               ;DOUBLE PRECISON ADD
        LDAX            D               ;LOAD 3RD NUMBER
        DCX             D               ;DECREMENT POINTER
        MOV             C,A             ;PUT 3RD NUMBER IN REG. C
        DAD             B               ;DOUBLE PRECISION ADD
        LDAX            D               ;LOAD 4TH NUMBER
        MOV             C,A             ;PUT 4TH NUMBER IN REG. C
        DAD             B               ;DOUBLE PRECISION ADD
        STC                             ;SET CARRY
        CMC                             ;COMPLEMENT CARRY, (CLEAR IT)
        MOV             A,H             ;MOVE HIGH ORDER BYTE OF
                                        ;RESULT INTO REGISTER A
        RAR                             ;DIVIDE UPPER BYTE BY TWO
        MOV             H,A             ;SWAP REG. H AND L
        MOV             A,L
        RAR                             ;DIVIDE LOWER BYTE BY TWO
        MOV             L,A             ;SWAP REG. H AND L
        MOV             A,H
        RAR                             ;DIVIDE UPPER BYTE BY TWO
        MOV             A,L             ;LOAD LOWER BYTE
        RAR                             ;DIVIDE BY TWO ANSWER IN ACC.
        DCX             D               ;DECREMENT POINTER
        STAX            D               ;STORE RESULT IN 5TH LOCATION
        HLT
        END
```

The preceding program calculates the average of four numbers and stores the result in a specified location. The program is assembled and emulation mode 0 is assigned. The absolute binary object code is read into program memory with the LOAD command. Entering the DEBUG command as follows places the system in debug mode.

```
>DEBUG
```

The program may now be traced for errors in execution flow.

To trace all instructions in the program's execution sequence, enter the command sequence below.

```
>TRACE ALL

>GO 0
```

The appropriate Trace lines are displayed, as follows:

```
LOC     INST     MNEM    OPER     SP     RF  RA  RB  RC  RD  RE  RH  RL
0000    AF       XRA     A        0000   46  00  00  00  00  00  00  00
0001    47       MOV     B,A      0000   46  00  00  00  00  00  00  00
0002    67       MOV     H,A      0000   46  00  00  00  00  00  00  00
0003    11FF13   LXI     D,13FFH  0000   46  00  00  00  13  FF  00  00
0006    1A       LDAX    D        0000   46  12  00  00  13  FF  00  00
0007    1D       DCX     D        0000   46  12  00  00  13  FE  00  00
.       .        .       .        .      .   .   .   .   .   .   .   .
.       .        .       .        .      .   .   .   .   .   .   .   .
.       .        .       .        .      .   .   .   .   .   .   .   .
```

Trace lines of all instructions are continuously displayed until a trace line termination condition is met.

8085A

The 8085A display includes all the information in the 8080A display, plus a display of the hexadecimal representation of the interrupt mask register as read by the 8085A RIM instruction. (See Appendix C for RIM format.) A binary representation of the current value of the SOD (Serial Output Data) bit is also provided.

```
LOC     INST     MNEM    OPER     SP     RF  RA  RB  RC  RD  RE  RH  RL  IM  SOD
0000    00       NOP              0000   02  00  00  00  00  00  00  00  07  0
0001    C30000   JMP     0000     0000   02  00  00  00  00  00  00  00  07  0
```

Syntax

DSTAT

PURPOSE

The DSTAT command causes display of the current debugging session status.

EXPLANATION

The DSTAT command sends a display line to the Debug display device. The DSTAT display line for a 8080A program takes the form shown below.

```
PC=xxxx  BP=xxxx RW  xxxx RW  M=x  SP=xxxx  RF=xx  RA=xx xx xx xx xx xx xx
```

All DSTAT display line values are in hexadecimal format. A description of the display line for a program written in 8080A assemble language follows.

PC The emulator processor's next instruction address.

BP The two possible active breakpoints and their parameters. If the R parameter is shown, a breakpoint is set to occur whenever an attempt is made to read from a specified address. If the W parameter is shown, a breakpoint is set to occur whenever an attempt is made to write to a specified address. If the RW parameter is shown, a breakpoint is set to occur whenever an attempt is made to read from or write to the specified address.

M The current emulation mode.

SP The value of the stack pointer (two bytes).

RF The value of the flag register.

RA The value of register A, followed by the values for the unlabeled registers B, C, D, E, H, and L.

EXAMPLE

Suppose breakpoints are set at addresses ØØØ9 and ØØØA in an 8080A program. Whenever an attempt is made to read (specified by "R") from either of these addresses, a breakpoint is set to occur. The following command lines set those breakpoints.

```
>BKPT 0009 R
>BKPT 000A R
```

When the program is executed with the GO command, the first breakpoint occurs at address 0009.

```
>GO

LOC     INST    MNEM    OPER    SP    RF  RA  RB  RC  RD  RE  RH  RL

0009    1A      LDAX    D       0000  92  00  00  00  13  FE  00  00
0009    BREAK

>
```

The second breakpoint occurs at address ØØØA:

```
>GO

000A    1D      DCX     C       0000  92  00  00  00  13  FD  00  00
000A    BREAK

>
```

A debug status line might now be useful to examine the current status of the debugging session.

```
>DSTAT

PC=000B  BP=0009  R  000A  R  M=0  SP=0000  RF=92 RA=00 00 00 13 FD 00 00
```

The debug status line displays the emulator processor's next instruction address (ØØØB), the active breakpoints and their parameters (ØØØ9 R and ØØØA R), the emulation mode (0), the stack pointer contents (0000), the flag register contents (92), the emulator processor register contents (00, 00, 00, 13, FD, 00, 00).

8085A

The display line for the current line in an 8085A program takes the form shown below:

```
LOC     INST    MNEM    OPER    SP    RF  RA  RB  RC  RD  RE  RH  RL  IM  SOD

0000    00      NOP             0000  02  00  00  00  00  00  00  00  07  0
0001    C30000  JMP     0000    0000  02  00  00  00  00  00  00  00  07  0

>DSTAT

PC=0004  BP=0001 R  M=0  SP=0000  RF=02  RA=00 00 00 00 00 00 00
SOD=0    SID=0   17=0   16=0   15=0   IE=0   M7=0   M6=0   M5=0
```

The values and format of the first line are identical to those of the 8080A display line.

In the 2nd line, all values are single digit binary.

SOD	Serial output data
SID	Serial input data
17, 16, 15	RST 7.5, RST 6.5, RST 5.5 interrupts pending status.
IE	Interrupt enable, enables INTR, RST 5.5, RST 6.5, RST 7.5.
M7, M6, M5	RST 7.5, RST 6.5, RST 5.5 interrupt masks (0 equals enable).

Syntax

SET {initial register} {first hex value} [second hex value] . . .

PURPOSE

To reassign hexadecimal values to the 8080A Emulator Processor two-byte stack pointer or other registers, enter the SET command line.

EXPLANATION

Values may be reassigned for a continuous series of one or more registers, beginning with the first register specified. This series should not exceed the available registers.

The 8080A Emulator Processor registers may be reassigned in the following sequence:

```
SP (two bytes) RF  RA  RB  RC  RD  RE  RH  RL
```

A definition of each element in the SET sequence follows:

SP Stack Pointer (two bytes)

RF Flag Register

RA Register A

RB Register B

RC Register C

RD Register D

RE Register E

RH Register H

RL Register L

Note that when reassigning hexadecimal values to the stack pointer register, a two-byte hexadecimal value must be specified for the register. When values under two bytes in length are specified for the stack pointer register, the high byte of the register is filled with zeros.

EXAMPLE

Suppose the stack pointer and register contents below are displayed by the DSTAT command:

```
PC=0009   BP=      M=0  SP=FDFA   RF=92   RA=00 00 00 13 13 FD 00
```

The following command line reassigns values to the stack pointer, the flag register, and registers A, B, and C.

```
>SET SP 1A 2A 22 01 02
```

Another look at the register contents with the DSTAT command shows the change.

```
>DSTAT

PC=0009  BP=     M=0  SP=001A  RF=2A  RA=22 01 02 13 13 FD 00
```

Since the stack pointer is a 16-bit (2-byte) register, and only eight bits (1A) were specified, the high byte is filled with zeros. To set the stack pointer with hexadecimal values exceeding one byte, enter both bytes as one value. For example:

```
        SET     SP  012A

>DSTAT

PC=009  BP=     M=0   SP=012A  RD=2A  RA=22 01 02 13 13 FD 00
```

8085A

The 8085A Emulator Processor registers may be reassigned in the following sequence:

```
I7, IE, M7, M6, SP, RF, A, B, C, D, E, H, L
```

(R7, M7, M6, and M5 are not names of actual registers, but may have binary value assigned to them as shown. IE may be set or reset.)

I7 resets (assigns a value of zero to) the highest priority interrupt, RST 7.5. Thus, the parameter following "I7" will be automatically read as zero, regardless of the value entered. The parameter must not be omitted, however. The I7 should be entered as follows:

SET {I} {0}

IE allows the interrupts to be enabled (1) or disabled (Ø).

M7, M6, and M5 allow a value of zero or one to be SET to the respective interrupt masks.

Suppose after invoking the DSTAT command, the following contents are observed:

```
PC=0000  BP=0001  WR  M=0  SP=0000  RF=02  RA=00 00 00 00 00 00 00
SOD=0    SID=0    17=0  16=0  15=0  IE=0  M7=1  M6=1  M5=1
```

You wish to assign new values to M7, M6, and register A. The SET command line that follows performs this function.

```
>SET M7,0,0,,,,2D
```

All parameters need not be present following the SET command, but the ones that are present must be placed in the correct sequence shown.

If the next parameter directly follows in sequence, it is not necessary to enter its name. In the example, "M6" is omitted, but the zero value holds its place. Commas may be inserted as delimiters for parameters not desired in the command line. The commas in the example delimit the parameter field for M5, SP, and F; the hexadecimal value "2D" is placed in register A.

Syntax

DISM [lower address] [upper address]

PURPOSE

The DISM command dis-assembles object code in memory back into Assembler mnemonics and hexadecimal operands.

EXPLANATION

The "lower address" parameter represents the program or user prototype memory address where disassembly begins. The default value is 0. The "upper address" parameter represents the program or user prototype memory address where disassembly ends.

The DISM command prints a display on the system console consisting of the address of an instruction, its hexadecimal object code, mnemonic, and operand. The display takes the form shown below.

LOC INST MNEM OPER

LOC is the memory address of the instruction.

INST is the hexadecimal object code at the address.

MNEM is the dis-assembled mnemonic of the instruction.

OPER is the hexadecimal operand(s), of the instruction.

See Appendix H for a complete list of dis-assembled 8080A/8085A instructions.

Syntax

RESET

PURPOSE

The RESET command changes the contents of certain 8085A emulator processor registers.

EXPLANATION

The registers changed are:

	The Registers Changed Are	Value is Changed To
PC	The Program Counter	0000
SOD	The Serial Output Data	0
I7	The RST 7.5 Interrupt Pending Status	0
IE	The Interrupt Enable Flag	0
M7	RST 7.5 Interrupt Mask	1
M6	RST 6.5 Interrupt Mask	1
M5	RST 5.5 Interrupt Mask	1

EXAMPLE

Assume the following DSTAT display:

```
PC=010C  BP=0103 RW  0504 R  M=0  SP=FF8C  RF=54  RZ=07 FF FF FF FF 05 04
SOD=0   SID=0   17=0   16=0   15=0   IE=0   M7=1   M6=1   M5=1
```

The RESET command is then entered:

```
>RESET

>DSTAT

PC=0000  BP=0103 RW  0504 R  M=0  SP=FF8C  RF=00  RA=07 FF FF FF FF 05 04
SOD=0   SID=0   17=0   16=0   15=0   IE=0   M7=1   M6=1   M5=1
```

Section 13

8080A REAL-TIME-PROTOTYPE-ANALYZER

If the system is operating in emulation mode 1, and the prototype clock fails, the system may not recover. The System Restart switch must be toggled to make the system operational again.

Section 14

PROTOTYPE CONTROL PROBE

INTRODUCTION

The prototype control probe links the prototype hardware to the emulator processor module. The probe replaces the microprocessor on the prototype hardware, permitting the prototype to be tested and debugged under 8002A μProcessor Lab control.

Hardware debugging is accomplished through the emulator processor; the emulator software; and the probe. Programs written for execution by the microprocessor can be monitored completely, and emulation permits thorough prototype testing.

DESCRIPTION AND INSTALLATION

The prototype control probe consists of three connected parts: a 6-foot ground plane cable pair, a driver/receiver board, and an 18-inch cable pair with a 40-pin plug. The complete assembly is shown in Fig. 14-1.

The 6-foot ground plane cable pair for the 8080A prototype control probe consists of two 40-conductor flat cables with ground, power, signal lines, and ground plane attached to the 8002A chasis. The free end of the cable pair connects to the emulator processor module by means of two edge card connectors inserted at the top of the emulator board.

The 6-foot ground plane cable pair for the 8085A prototype control probe consists of two 40-conductor flat cables with ground, power, and signal lines. The free end of the cable pair connects to the emulator processor module by means of a cable terminaton card inserted at the top of the emulator board. The 8085A CPU is then moved to the prototype control probe module to minimize emulation delays.

Receivers for data, address, and circuit board control are located in the prototype control probe assembly. The probe assembly provides signal integrity and minimizes loading on circuits connected to the microprocessor socket.

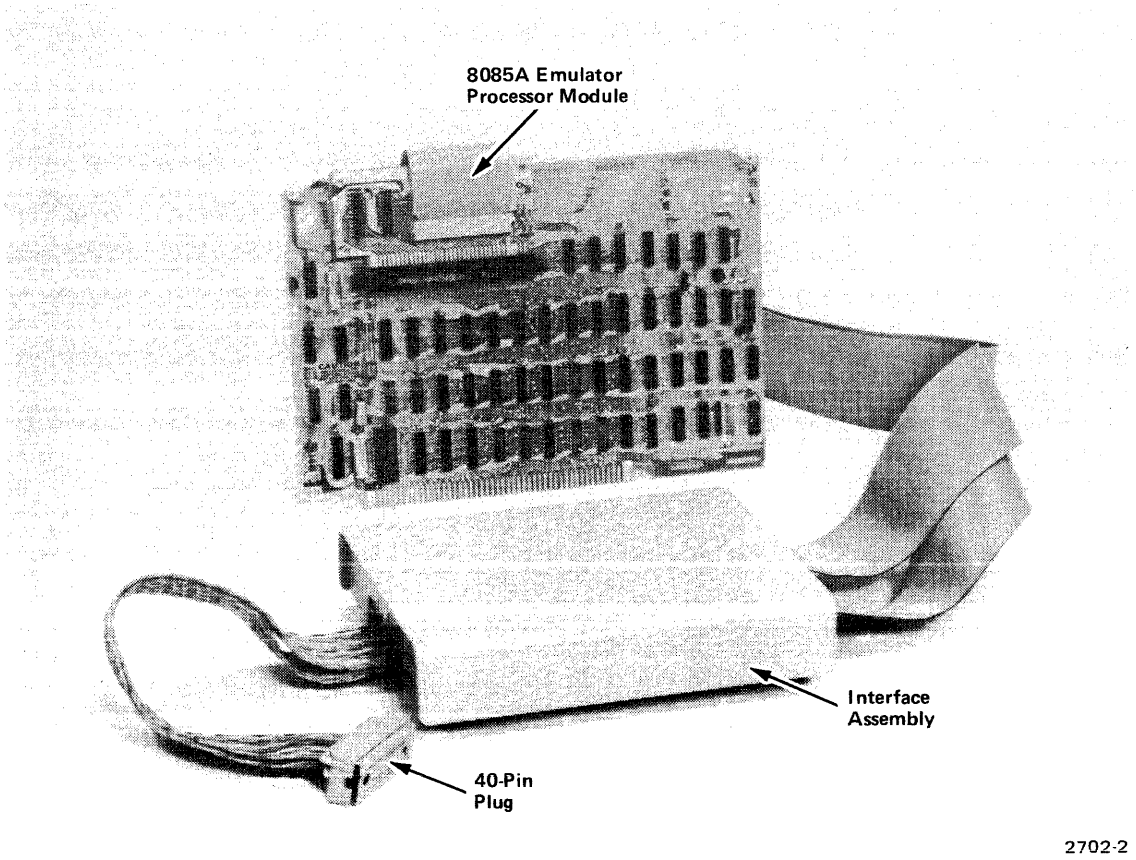


Fig. 14-1. 80805A Emulator processor and prototype control probe assembly.

A 40-pin plug at the end of the 18-inch twisted-pair cables fits into the prototype microprocessor socket. Pin 1 on the plug must be mated to receptacle 1 of the socket. An indentation is located near pin 1 on the plug to aid in pin identification. Refer to Fig. 14-2, demonstrating proper plug insertion.

CAUTION

If the plug is incorrectly inserted, damage to the prototype control probe will result. Fig. 14-2 illustrates the proper method for plug insertion.

If the plug is incorrectly inserted, the following parts may need to be replaced:

8080A Prototype Control Probe Driver Receiver Board

DIP No.	Tektronix Part No.	Manufacturer No.
U2010	156-0999-00	8T98
U2030	156-0996-00	8T26
U2040	156-0996-00	8T26
U2060	156-0996-00	8T26
U2080	156-0999-00	8T98
U2100	156-0999-00	8T98
U3100	156-0999-00	8T98
U3010	156-0323-00	74S04

8080A Personality Board

DIP No.	Tektronix Part No.	Manufacturer No.
U1020	156-0180-00	74S00
U2030	156-0180-00	74S00

8085A Prototype Control Probe Driver Receiver Board

DIP No.	Tektronix Part No.	Manufacturer No.
U1010	156-0956-00	74LS244
U3010	156-0956-00	74LS244
U6020	156-0928-00	74LS243
U6010	156-0928-00	74LS243
U2010	156-0956-00	74LS244
U5010	156-0382-00	74LS00
U5020	156-0382-00	74LS00

8085A Prototype Control Probe

DIP No.	Tektronix Part No.	Manufacturer No.
	156-0462-00	7414
→ Dual-Junction FET	151-1049-00	D/2N38 Fmly
Diode	152-0008-00	T12G

8080A Emulator Processor Board

3AG 250V 2A Fast Blow Fuse — Tektronix Part No. 159-0023-00 (DIP's are mounted in sockets for easy replacement.)

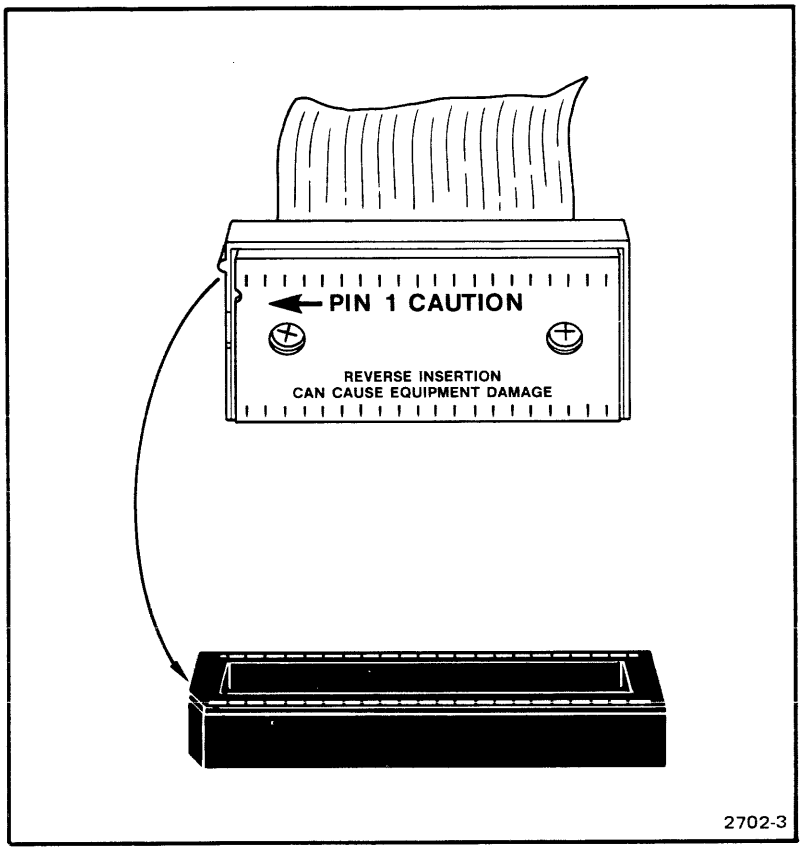


Fig. 14-2. Proper plug insertion.

When using the spring-plated-protected 40-pin plug with a zero-insertion-force socket, place the 40-pin low-profile DIP socket (included) between the plug and the socket.

The prototype control probe, properly installed, is shown in Fig. 14-3. If the pins on the plug are not shorted, the cable assembly can remain connected to the prototype hardware while the prototype control probe is not in use.

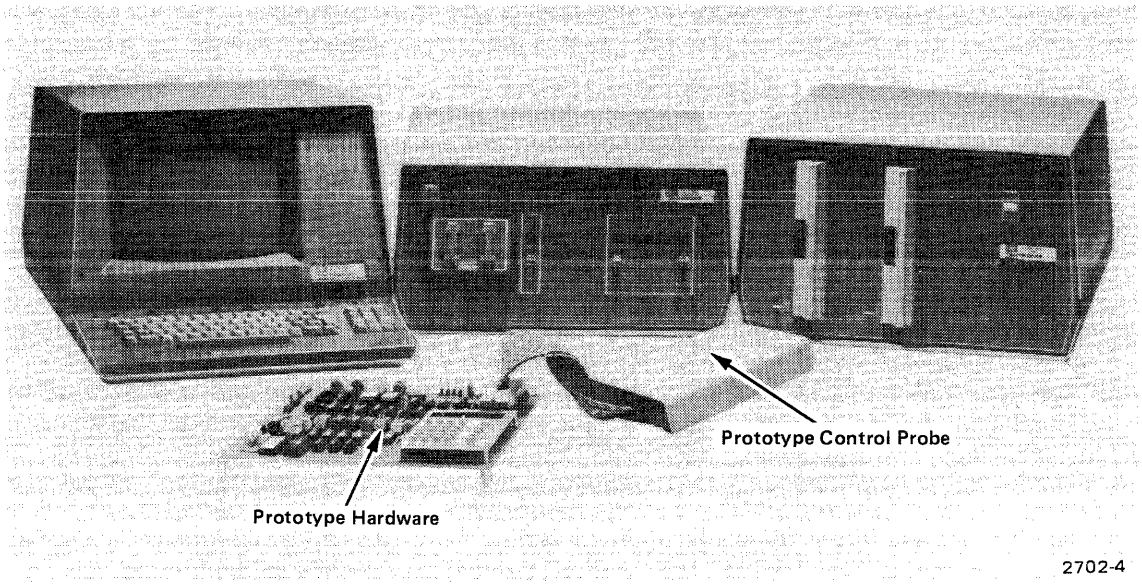


Fig. 14-3. Prototype control probe connected to prototype hardware.

OPERATION

Once the prototype control probe is connected to the prototype hardware, the prototype hardware and software are exercised under TEKDOS control. Refer to the 8002A μProcessor Lab System User's Manual for details.

Section 15

8080A CONVERTER

INTRODUCTION

This section describes the TEKDOS syntax required to convert Intel source code into Tektronix source code.

Syntax

CONVERT { source file [/disc drive] / source device } { destination file [/disc drive] / destination device }

PURPOSE

The CONVERT command invokes the Converter when the 8002A μProcessor Lab is under TEKDOS control.

EXPLANATION

The Tektronix 8080A Converter is a system program that translates 8080A programs from Intel's 8080 Macro Assembler format to Tektronix 8080A/8085A Assembler, Version 3, format. The Intel 8080A Macro Assembler generates absolute object code that is described in the Intel 8080A Assembly Language Programming Manual (1976). The converter input is source code written in Intel assembly language. The Converter output is source code written in Tektronix assembly language.

Both the source and destination parameters are mandatory. The first parameter is the name of the file or device from which the Converter reads the Intel assembly language source code. The second parameter is the name of the destination file or device to which the Converter outputs the converted Tektronix assembly language code.

The Tektronix 8080A Converter output is then acceptable as input to the Tektronix 8080A Assembler. It is important to note that incompatibilities exist between the Intel and the Tektronix assembly language formats. They are discussed in the subsection entitled "Incompatibilities". Often, however, the user will be able to assemble the converted code without actually having to rewrite any code.

The end of the conversion is signaled by a console message. This message also indicates the number of replacements (see subsection on Replacements) made and the number of incompatibility warnings given.

The Tektronix 8080A Converter makes two passes through the source program. During the first pass a partial symbol table is constructed for use in converting macros. During the second pass the converted program is generated.

Input to the Converter may come from any valid TEKDOS input device. However, since the Converter makes two passes through the source material, the same material should be available for both passes. The Converter terminates after reading to the end of the input source file on the second pass. The converted program may be output to either a flexible disc file or to a device.

REPLACEMENTS

Each Intel assembly language statement consists of four fields in the following order:

LABEL FIELD OPERATION FIELD OPERAND FIELD COMMENT FIELD

In conversion to Tektronix code, replacements are made in all fields.

In the label field the following replacements are made:

1. A colon (:) after a label, or two consecutive colons after a label, is replaced by a blank.
2. In a MACRO directive label field, the macro name is replaced by a blank. The macro name is placed in the operand field of this statement.
3. Any local symbol, as defined by an EQU directive or by its use in the label field of a macro definition block statement, is replaced by the concatenation of the symbol with the unique macro call number character, @, enclosed in substitution delimiters (single quotes).

In the operation field the following replacements are made:

1. The DB directive is replaced by either the BYTE or the ASCII directive. The Converter creates one BYTE statement for each numeric operand and one ASCII statement for each string operand.
2. The DW directive is replaced by the WORD directive.
3. The DS directive is replaced by the BLOCK directive.

In the operand field, the following replacements are made:

1. The first single quote (') on a MACRO call is replaced by a left square bracket, [.
2. The last single quote (') on a MACRO call is replaced by a right square bracket,] .
3. Any single quotes between the first and single quotes on a MACRO call are treated as for strings. (See items 4 and 5 which follow).

4. On all statements except MACRO directives, the first and last single quotes are each replaced by double quotes (").

5. Any two consecutive apostrophes that follow the first single quote, are replaced by an up-arrow (↑) or caret (^) followed by the single quote (').

6. Any double quote (") that follows the first single quote is replaced by an up-arrow (↑) or caret (^) followed by the double quotation mark.

7. Any up-arrow (↑) or caret (^) that follows the first single quote is replaced by two up-arrows or carets.

8. Any local symbol (as defined by an EQU directive or by its use in the label field of a macro definition block statement) in the operand field of a macro definition block statement is replaced by the concatenation of the symbol with the unique macro call number character, @, enclosed in substitution delimiters (single quotes).

9. All macro parameters in the operand field of a MACRO directive are replaced by the macro name from the label field.

10. In all other macro definition block statements, any macro parameter is replaced by its position number enclosed in single quotes. For example, the following Intel format macro definition statement indicates that parameter X has position number one and parameter Y has position number two:

```
SHV     MACRO     X,Y
```

    Therefore, in the operand field of a subsequent statement within this macro definition block, any X is replaced by '001', and any Y is replaced by '002'.

11. Intel logical operators (NOT, AND, OR, and XOR) are replaced by Tektronix logical operators (\, &, !, and !!).

In the comment field any single quote is replaced by an up-arrow followed by a single quote (↑'), and any up-arrow is replaced by two up-arrows (↑↑).

Symbol replacements are made in the label, operation, and operand fields. An illegal symbol, one which contains one or more at-signs (@) or question marks (?), is replaced by a converted symbol. This converted symbol is formed by replacing any initial @ by A$, any initial ? by Q_, any subsequent @ by $, and any subsequent ? by _.

A symbol reserved by the Tektronix Assembler cannot be defined by the user. The following symbols are reserved by the Tektronix Assembler, although they are not reserved in the Intel Macro Assembler:

ASCII	ENDR	PAGE
BASE	EXITM	RIM
BLOCK	HI	SEG
BYTE	LIST	SIM
CND	LO	SPACE
CON	ME	SYM
DEF	MEG	TRM
ELSE	NAME	WORD
ENDOF	NCHR	XREF

These reserved symbols are replaced by the same symbol followed by "$". Any statement that contains a label, no operation, and no operand is replaced by the label followed by the EQU directive and the location counter character, as shown in the following example:

```
LABEL   OPERATION       OPERAND         COMMENT

Intel Code:

SRBUF                                   ;DEFINE BUFFER

Tektronix Converted Code:

SRBUF   EQU             $               ;DEFINE BUFFER
```

INCOMPATIBILITIES

There are several important differences between the assembly process of the Intel Macro Assembler and the Tektronix Assembler. These differences do not involve easily recognized syntax variations, but rather the actual operation of the assemblers. It is highly recommended that the user evaluate the converted program carefully before assembly to ensure that the program will assemble as intended. The following differences are important to keep in mind when evaluating a program.

Arithmetic

Both Assemblers use two's complement arithmetic. The Intel Macro Assembler treats all arguments of its expression operators as unsigned quantities; the Tektronix Assembler treats them as signed.

Expression Operators

The types of expression operators used by the two Assemblers differ, as does the relative precedence of these operator types. Therefore, an expression is not necessarily evaluated identically by the Intel Macro Assembler and the Tektronix Assembler.

The following table illustrates the precedence of the expression operators in the Intel Macro Assembler. All items on the same line have equal precedence.

Highest Priority	*	/	MOD SHL SHR
	+	-	(Unary and Binary)
	NOT		
	AND		
Lowest Priority	OR	XOR	

The following table illustrates the precedence of the expression operators and functions in the Tektronix Assembler. All items on the same line have equal precedence.

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

It is strongly recommended that the user check expressions to ensure that they assemble as intended. The Converter does not replace an Intel expression with a Tektronix expression in which precedence has been adjusted appropriately. However, an incompatibility warning is given for NOT operator conflicts. (See the subsection, "Incompatibility Warnings".)

Macro Call Parameters

The Intel Macro Assembler passes parameters by value. The Tektronix Assembler passes parameters by name. Therefore, a macro may not expand to the intended code. The following example illustrates this incompatibility:

```
        INTEL CODE                              TEKTRONIX CONVERTED CODE

        SAMPL   MACRO   X                               MACRO   SAMPL
        VAL     SET     4               VAL             SET     4       MACRO
                DW      X                               WORD    '001'   DEFINITION
                ENDM                                    ENDM

        VAL     SET     0               VAL             SET     0       PROGRAM
                                                                        STATEMENT

                SAMPL   VAL                             SAMPL   VAL     MACRO CALL

+       VAL     SET     4       +       VAL             SET     4       MACRO
+               DW      0       +                       WORD    VAL     EXPANSION
```

In this example, the Intel Macro Assembler limits macro parameters to the operand field. However, the Tektronix Assembler allows macro parameters in all statement fields.

Single quote conversion within a macro parameter may produce code that does not assemble in the intended manner. All macro definitions should be carefully reviewed after conversion to ensure correct assembly. To remind the user of potential conflicts, an incompatibility warning is given for each MACRO directive.

Scope of Symbols

In Intel assembly language, any SET symbol defined in a macro expansion statement is local to the current macro expansion, unless that symbol has previously been defined globally by another SET statement. In Tektronix assembly language, unique labels for macro expansion can be generated by using a label concatenated with the unique macro call number character, @, enclosed in single quotes.

String Capability

The Tektronix Assembler has a powerful string variable capability, including string function operators. These string features are particularly useful in macros.

Incompatibility Warnings

The converter includes incompatibility warnings with the converted output code in the form of WARNING directive statements. The following types of warnings are possible:

1. ILLEGAL OPERAND. An instruction mnemonic is used in the operand field.

2. OPERATOR PRECEDENCE CONFLICT. A NOT operator is used in an expression containing one or more of the following operators: +, —, *, /, MOD, SHL, SHR.

3. MACRO INCOMPATIBILITY. See the previous discussion of macro call parameters for potential incompatibilities.

4. NEXT LINE CONTAINS SOURCE FORMAT ERROR. An incorrect Intel source line follows.

The incompatibility warning statement precedes the line containing the incompatibility. Only one incompatibility warning may precede a converted source line. The following is an example of a source line with an incompatibility;

```
LABEL   OPERATION       OPERAND

TBL:    DB              (ADD C)
```

The converted code is output as follows:

```
LABEL   OPERATION       OPERAND         COMMENT

        WARNING                         ; ILLEGAL OPERAND
TBL     BYTE            (ADDC)
```

OUTPUT FORMAT

The destination file or device receives the converted code. Converted code consists of: statements copied unchanged from the Intel source code, corrected source code with appropriate replacements, and incorrect statements preceded by incompatibility warnings.

Here is an example of an Intel assembly language code segment:

```
LABEL   OPERATION       OPERAND

        XRA             A
VAL     EQU             100H
        LXI             H, VAL
        MOV             B, M
@COMP:  CMP             B
        JZ              DEF
        INR             B
        JMP             @COMP
DEF:    DB              VAL, 'A'
```

Here is that same code, after conversion:

```
LABEL   OPERATION                       OPERAND

; XXXX  TEKTRONIX 8080A CONVERTER       V1.0
        XRA                             A
VAL     EQU                             100H
        LXI                             H, VAL
        MOV                             B, M
A$COMP  CMP                             B
        JZ                              DEF$
        INR                             B
        JMP                             A$COMP
DEF$    BYTE                            VAL
        BYTE                            "A"
; XXXX CONVERSION COMPLETE
; XXXX NUMBER OF WARNINGS=0
; XXXX NUMBER OF REPLACEMENTS=9
```

At the beginning of the conversion a message is sent to the console. This message indicates the type of microprocessor conversion program and the version designation.

At the end of the conversion a message is sent to the console. This message indicates that the conversion is complete, and shows the number of warnings, and the number of replacements.

Appendix A

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
Ø . . . 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
&	binary logical operator, and
!	binary logical operator, inclusive or
!!	binary logical operator, exclusive or
SPACE	field delimiter

SYMBOLS	DEFINITION
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

Appendix B

ASSEMBLER DIRECTIVES

DIRECTIVE	OPERATION
ASCII	stores ASCII text in memory
BLOCK	reserves a specified number of bytes in memory
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
PAGE	begins the next listing line on the following page
REPEAT	enables macro lines between REPEAT and ENDR directives to be assembled repeatedly

(Directives continued on next page)

DIRECTIVE	OPERATION
RESERVE	sets aside a workspace in memory
RESUME	continues definition of code for a given section
SECTION	declares Linker section, assigns name, defines parameters
SET	assigns or reassigns an expression value to a string or numeric variable symbol
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]	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] [,DBG]	[;charstring]
[symbol]	MACRO	{ symbol }	[;charstring]
[symbol]	NAME	{ symbol }	[;charstring]
[symbol]	NOLIST	[CND] [,TRM] [,SYM] [,CON] [,MEG] [,ME] [,DBG]	[;charstring]
[symbol]	ORG	{[/] expression}	[;charstring]
[symbol]	PAGE		[;charstring]
[symbol]	REPEAT	{ expression1 } [,expression2]	[;charstring]
[symbol]	RESERVE	{ symbol, expression } [,PAGE / ,INPAGE]	[;charstring]
[symbol]	RESUME	[symbol]	[;charstring]
[symbol]	SECTION	{ symbol } [,PAGE / ,INPAGE / ,ABSOLUTE]	[;charstring]
{symbol}	SET	{ expression }	[;charstring]
[symbol]	SPACE	[expression]	[;charstring]

(Directives continued on next page)

LABEL	OPERATION	OPERAND	COMMENT
[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

SUMMARY OF 8080/8085 INSTRUCTIONS

All 8080A/8085A instructions are summarized in this appendix. For a detailed description of the instruction set, consult an 8080/8085 assembly language programming manual.

Each 8080A/8085A instruction consists of an operation code and up to two operands.

An operation involving implied operands consists of an operation code only. Other instructions require one or two written operands, dependent upon the operation to be performed.

Descriptive symbols used in this appendix to represent items in the operand field are as follows:

Symbol	Description
r, rl, r2	One of the registers A, B, C, D, E, H, L or an expression that evaluates to a numeric register value assigned by the Assembler as shown in this table:

register	value
B	Ø
C	1
D	2
E	3
H	4
L	5
A	7

pair — One of the register pairs:

B represents B and C.
D represents D and E.
H represents H and L.
SP represents a two-byte stack pointer register.

pairh	High register of a register pair: B, D, H, or SPH.
pairl	Low register of a register pair: C, E, L, or SPL.
exp8	An expression representing a one byte data or address constant or I/O port address.
exp16	An expression representing a two byte data or address constant.
exp16h	The high byte of a two byte constant.
exp16l	The low byte of a two byte constant.
expv	An expression representing a restart vector number. Its possible values are Ø to 7.
M	A memory reference to the contents of the address contained in the H, L register pair. The H register contains the high byte. The L register contains the low byte. The assigned value of M is 6.
PSW	Two bytes containing register A and the state of the condition flags.
AØ	Bit Ø of register A.
A1	Bit 1 of register A.
A2	Bit 2 of register A.
A3	Bit 3 of register A.
A4	Bit 4 of register A.
A5	Bit 5 of register A.
A6	Bit 6 of register A.
A7	Bit 7 of register A.
SP	The two byte stack pointer.
PC	The two byte program counter in the CPU (points to current instruction).

PCH	High byte of the program counter.
PCL	Low byte of the program counter.
F	Condition flags, treated as an 8-bit register S Z Ø HCY Ø P 1 CY
←	Left arrow indicates "is transferred to".
←→	Indicates "is exchanged with".
*	Multiplication operator.
\	Logical NOT operator.
&	Logical AND operator.
!	Logical Inclusive OR operator.
!!	Logical Exclusive OR operator.
( )	Refers to contents of address, register, or flag.
(( ))	Refers to the contents of a location whose address is contained in the specified register (indirect addressing).

Condition Codes

CY	Carry-borrow flag (from bit 7), 1 if the result of an instruction is a carry or borrow.
P	Parity flag, 1 if the result of an instruction has an even parity.
HCY	Half carry flag (from bit 3), 1 is the result of an instruction has a half carry.
Z	Zero flag, 1 if the result of an instruction is zero.
S	Sign flag, 1 if the result of an instruction is negative.

Condition Code Indicators

x	The effect on the flag is dependent on the result of an instruction.
Ø	Reset
u	The flag is unaffected by the result of an instruction.
1	Set

Source Module Syntax		Object Module	Machine	Instruction Description	Condition Codes				
Operation	Operand	Bytes	Cycles		CY	P	HCY	Z	S
DATA TRANSFER INSTRUCTIONS									
LDA	exp16	3	4	(A)←(exp16)	u	u	u	u	u
LDAX	pair	1	2	(A)←((pair)) Note: only pair = B or pair = D is allowed.	u	u	u	u	u
LHLD	exp16	3	5	(L)←(exp16) (H)←(exp16 + 1)	u	u	u	u	u
LXI	pair,exp16	3	3	(pairh)←exp16h (pairl)←exp16i	u	u	u	u	u
MOV	r1,r2	1	1	(r1)←(r2)	u	u	u	u	u
MOV	r,M	1	2	(r)←(M)	u	u	u	u	u
MOV	M,r	1	2	(M)←(r)	u	u	u	u	u
MVI	r,exp8	2	2	(r)←exp8	u	u	u	u	u
MVI	M,exp8	2	3	(M)←exp8	u	u	u	u	u
SHLD	exp	3	5	(exp16)←(L) (exp 16 + 1) ←(H)	u	u	u	u	u
STA	exp16	3	4	(exp16)←(A)	u	u	u	u	u
STAX	pair	1	2	((pair))←(A) Note: only pair = B or pair = D is allowed.	u	u	u	u	u
XCHG		1	1	(H)↔(D)	u	u	u	u	u
				(L)↔(E)	u	u	u	u	u
REGISTER INCREMENT AND DECREMENT INSTRUCTIONS									
DCR	M	1	3	(M)←(M) − 1	u	x	x	x	x
DCR	r	1	1	(r)←(r) − 1	u	x	x	x	x
DCX	pair	1	1	(pair)←(pair) − 1	u	u	u	u	u
INR	M	1	3	(M)←(M) + 1	u	x	x	x	x
INR	r	1	1	(r)←(r) + 1	u	x	x	x	x
INX	pair	1	1	(pair)←(pair) + 1	u	u	u	u	u

Source Module Syntax		Object Module	Machine	Instruction Description	Condition Codes				
Operation	Operand	Bytes	Cycles		CY	P	HCY	Z	S
ARITHMETIC INSTRUCTIONS									
ACI	exp8	2	2	(A)←(A) + exp8 + (CY)	x	x	x	x	x
ADC	M	1	2	(A)←(A) + (M) + (CY)	x	x	x	x	x
ADC	r	1	1	(A)←(A) + (r) + (CY)	x	x	x	x	x
ADD	M	1	2	(A)←(A) + (M)	x	x	x	x	x
ADD	r	1	1	(A)←(A) + (r)	x	x	x	x	x
ADI	exp8	2	2	(A)←(A) + exp8	x	x	x	x	x
CMP	r	1	1	(A) – (r) Note: The result is not stored. Z is set to 1 if (A) = (r). CY is set to 1 if (A) < (r)	x	x	x	x	x
CMP	M	1	2	(A) – (M) Note: The result is not stored. A is set to 1 if (A) = (M). CY is set to 1 if (A) < (M).	x	x	x	x	x
CPI	exp8	2	2	(A) – exp8 Note: The result is not stored. Z is set to 1 if (A) = exp8. CY is set to 1 if (A) < exp8.	x	x	x	x	x
DAA		1	1	Decimal adjust register A, accumulator to form two four-bit Binary-Coded-Decimal digits. If D_L is greater than 9 or HCY is set ADD 6 to ACC Then If D_H is greater than 9 or CY is set ADD 60H to ACC. This is only valid after an ADD operation. (Non-Intel-manufactured cpu will perform the operation differently.)	x	x	x	x	x
DAD	pair	1	3	(H,L)←(H,L) + (pair)	x	u	u	u	u
SBI	exp8	2	2	(A)←(A) – exp8 – (CY)	x	x	x	x	x
SBB	M	1	2	(A)←(A) – (M) – (CY)	x	x	x	x	x
SBB	r	1	1	(A)←(A) – (M) – (CY)	x	x	x	x	x
SUB	M	1	2	(A)←(A) – (M)	x	x	x	x	x
SUB	r	1	1	(A)←(A) – (r)	x	x	x	x	x

Source Module Syntax		Object Module	Machine	Instruction Description	Condition Codes				
Operation	Operand	Bytes	Cycles		CY	P	HCY	Z	S
SUI	exp8	2	2	(A)←(A) – exp8	x	x	x	x	x
LOGICAL INSTRUCTIONS									
ANA	M	1	2	(A)←(A) & (M)	Ø	x	x	x	x
ANA	r	1	1	(A)←(A) & (r)	Ø	x	x	x	x
ANI	exp8	2	2	(A)←(A) & exp8	Ø	x	Ø	x	x
CMA		1	1	(A)←\(A)	u	u	u	u	u
CMC		1	1	(CY)←\(CY)	x	u	u	u	u
ORA	M	1	2	(A)←(A) ! (M)	Ø	x	Ø	x	x
ORA	r	1	1	(A)←(A) ! (r)	Ø	x	Ø	x	x
ORI	exp8	2	2	(A)←(A) ! exp8	Ø	x	Ø	x	x
STC		1	1	(CY)←1	1	u	u	u	u
XRA	M	1	2	(A)←(A) !! (M)	Ø	x	Ø	x	x
XRA	r	1	1	(A)←(A) !! (r)	Ø	x	Ø	x	x
XRI	exp8	2	2	(A)←(A) !! exp8	Ø	x	Ø	x	x
ROTATE INSTRUCTIONS									
RAL		1	1	(CY)←(A7) (A7 to A1)←(A6 to AØ) (AØ)←(CY)	x	u	u	u	u
RAR		1	1	(CY)←(AØ) (A6 to AØ)←(A7 to A1) (A7)←(CY)	x	u	u	u	u
RLC		1	1	(A7 to A1)←(A6 to AØ) (AØ)←(A7) (CY)←(A7)	x	u	u	u	u
RRC		1	1	(A6 to AØ)←(A7 to A1) (A7)←(AØ) (CY)←(AØ)	x	u	u	u	u
BRANCH INSTRUCTIONS									
CALL	exp16	3	5	((SP) – 1)←(PCH) ((SP) – 2)←(PCL) (SP)←(SP) – 2 (PC)←exp16	u	u	u	u	u

Source Module Syntax		Object Module	Machine	Instruction Description	Condition Codes				
Operation	Operand	Bytes	Cycles		CY	P	HCY	Z	S
BRANCH INSTRUCTIONS (contd.)									
CC	exp16	3	3 or 5	If (CY) = 1 then ((SP) – 1)←(PCH) ((SP) – 2)←(PCL) (SP)←(SP) – 2 (PC)←exp16 Note: Cycles = 5 if (CY) = 1 otherwise cycles = 3	u	u	u	u	u
CM	exp 16	3	3 or 5	If (S) = 1 then ((SP) – 1)←(PCH) ((SP) – 2)←(PCL) (SP)←(SP) –2 (PC)←exp16 Note: Cycles = 5 if (S) = 1, otherwise cycles = 3.	u	u	u	u	u
CNC	exp16	3	3 or 3	If (CY) = Ø then ((SP) – 1)←(PCH) ((SP) – 2)←(PCL) (SP)←(SP) – 2 (PC)←exp16 Note: Cycles = 5 if (CY) = Ø otherwise cycles = 3.	u	u	u	u	u
CNZ	exp16	3	3 or 5	If (Z) = Ø then ((SP) – 1)←(PCH) ((SP) – 2)←(PCH) (SP)←(SP) – 2 (PC)←exp16 Note: Cycles = 5 if (Z) = Ø, otherwise cycles = 3.	u	u	u	u	u
CP	exp16	3	3 or 5	If (S) = Ø then ((SP) – 1)←(PCH) ((SP) – 2)←(PCL) (SP)←(SP) – 2 (PC)←exp16 Note: Cycles = 5 if (S) = Ø, otherwise cycles = 3.	u	u	u	u	u
CPE	exp16	3	3 or 5	IF (P) = 1 then ((SP) – 1)←(PCH)	u	u	u	u	u

Source Module Syntax		Object Module	Machine	Instruction Description	Condition Codes				
Operation	Operand	Bytes	Cycles		CY	P	HCY	Z	S
BRANCH INSTRUCTIONS (contd.)									
				((SP) – 2)←(PCL) (SP)←(SP) – 2 (PC)←exp16 Note: Cycles = 5 if (P) = 1, otherwise cycles = 3.	u	u	u	u	u
CPO	exp16	3	3 or 5	If (P) = Ø then ((SP) – 1)←(PCH) ((SP) – 2)←(PCL) (SP)←(SP) – 2 (PC)←exp16 Note: Cycles = 5 if (P) = Ø, otherwise cycles = 3.	u	u	u	u	u
CZ	exp16	3	3 or 5	If (Z)= 1 then ((SP) – 1)←(PCH) ((SP) – 2)←(PCL) (SP)←(SP) – 2 (PC)←exp16 Note: Cycles = 5 if (Z) = 1, otherwise cycles 3.	u	u	u	u	u
JC	exp16	3	3	If (CY) = 1 then (PC)←exp16	u	u	u	u	u
JM	exp16	3	3	If (S) = 1 then (PC)←exp16	u	u	u	u	u
JMP	exp16	3	3	(PC)←exp16	u	u	u	u	u
JNC	exp16	3	3	If (CY) = Ø then (PC)←exp16	u	u	u	u	u
JNZ	exp16	3	3	If (Z) = Ø then (PC)←exp16	u	u	u	u	u
JP	exp16	3	3	If (S) = Ø then (PC)←exp16	u	u	u	u	u
JPE	exp16	3	3	If (P) = 1 then (PC)←exp16	u	u	u	u	u
JPO	exp16	3	3	If (P) = Ø then (PC)←exp16	u	u	u	u	u
JZ	exp16	3	3	If (Z) = 1 then (PC)←exp16	u	u	u	u	u
PCHL		1	1	(PCH)←(H) (PCL)←(L)	u	u	u	u	u
RC		1	1 or 3	If (CY) = 1 then (PCL)←((SP)) (PCH)←((SP)) +1) (SP)←(SP) + 2 Note: Cycles = 3 if (CY) = 1, otherwise cycles = 1.	u	u	u	u	u

Source Module Syntax Operation	Operand	Object Module Bytes	Machine Cycles	Instruction Description	Condition Codes CY	P	HCY	Z	S
BRANCH INSTRUCTIONS (contd.)									
RET		1	3	(PCL)←((SP)) (PCH)←((SP) + 1) (SP)←(SP) + 2	u	u	u	u	u
RM		1	1 or 3	If (S) = 1 then (PCL)←((SP)) (PCH)←((SP + 1) (SP)←(SP) + 2 Note: Cycles = 3 if (S) = 1, otherwise cycles = 1.	u	u	u	u	u
RNC		1	1 or 3	If (CY) = Ø then (PCL)←((SP)) (PCH)←((SP + 1) (SP)←(SP) + 2 Note: Cycles = 3 if (CY) = Ø, otherwise Cycles = 1.	u	u	u	u	u
RNZ		1	1 or 3	If (Z) = Ø then (PCL)←((SP)) (PCH)←((SP) + 1) (SP)←(SP) + 2 Note: Cycles = 3 if (Z) = Ø, otherwise Cycles = 1.	u	u	u	u	u
RP		1	1 or 3	If (S) = Ø then (PCL)←((SP)) (PCH)←((SP) + 1) (SP)←(SP) + 2 Note: Cycles = 3 if (S) = Ø, otherwise Cycles = 1.	u	u	u	u	u
RPE		1	1 or 3	If (P) = 1 then (PCL)←((SP)) (PCH)←((SP) + 1) (SP)←(SP) + 2 Note: Cycles = 3 if (P) = 1, otherwise Cycles = 1.	u	u	u	u	u

Source Module Syntax		Object Module	Machine	Instruction Description	Condition Codes				
Operation	Operand	Bytes	Cycles		CY	P	HCY	Z	S
BRANCH INSTRUCTIONS (contd.)									
RPO		1	1 or 3	If (P) = Ø then (PCL)←((SP)) (PCH)←((SP) + 1) (SP)←(SP) + 2 Note: Cycles = 3 if (P) = Ø, otherwise Cycles = 1.	u	u	u	u	u
RST	expv	1	3	((SP) – 1)←(PCH) ((SP) – 2)←(PCL) (SP)←(SP) – 2 (PC)←8 * expv	u	u	u	u	u
RZ		1	1 or 3	If (Z) = 1 then (PCL)←((SP)) (PCH)←((SP) + 1) (SP)←(SP) + 2 Note: Cycles = 3 if (Z) = 1, otherwise Cycles = 1	u	u	u	u	u
STACK INSTRUCTIONS									
POP	pair	1	3	(pairl)←((SP)) (pairh)←((SP) + 1) (SP)←(SP) + 2 Note: pair = SP is not allowed.	u	u	u	u	u
POP	PSW	1	3	(F)←((SP)) (A)←((SP) + 1) (SP)←(SP) + 2	x	x	x	x	x
PUSH	pair	1	3	((SP) – 1)←(pairh) ((SP) – 2)←(pairl) (SP)←(SP) – 2 Note: pair = SP is not allowed.	u	u	u	u	u
PUSH	PSW	1	3	((SP) – 1)←(A) ((SP) – 2)←(F) (SP)←(SP) – 2	u	u	u	u	u
SPHL		1	1	(SPH)←(H) (SPL)←(L)	u	u	u	u	u
XTHL		1	5	(L)←→((SP)) (H)←→((SP) + 1)	u	u	u	u	u

Source Module Syntax		Object Module	Machine	Instruction Description	Condition Codes				
Operation	Operand	Bytes	Cycles		CY	P	HCY	Z	S
INPUT/OUTPUT INSTRUCTIONS									
IN	exp8	2	3	(A)←(exp8) Note: exp8 = specified port.	u	u	u	u	u
OUT	exp8	2	3	(exp8)←(A) Note: exp8 = specified port.	u	u	u	u	u
INTERRUPT AND CONTROL INSTRUCTIONS									
DI		1	1	Disable interrupt after execution of next instruction.	u	u	u	u	u
EI		1	1	Enable interrupt after execution of next instruction. Therefore, the next instruction should not be a HLT.	u	u	u	u	u
HLT		1	1	Halt Processor.	u	u	u	u	u
NOP		1	4	No operation PC←PC + 1	u	u	u	u	u
*RIM:		1	1	Load accumulator with restart interrupt masks, any pending interrupts, and contents of serial input data line.	u	u	u	u	u
*SIM:		1	1	Use the contents of the accumulator to program the restart interrupt mask.					

*Available only on 8Ø85

Appendix D

SERVICE CALL FUNCTION CODES

CODE	FUNCTION
Ø1	Read ASCII and wait
Ø2	Write ASCII and wait
Ø3	Close device or file on channel
Ø4	Rewind file on channel
Ø5	Delete file on channel
Ø6	Rename file on channel
1Ø	Assign channel to device or channel
11	Get time (milliseconds)
13	Get parameter (procedure parameter buffer)
14	Get device type
15	Get device status
16	Get last console input character
17	Load overlay
18	Execute overlay
19	Suspend execution
1A	Exit
1C	Get parameter (emulation parameter buffer)
1F	Abort
21	Read ASCII and wait without echo from CONI.
41	Read binary and wait
42	Write binary and wait
57	Load overlay with bias
81	Read ASCII and proceed
82	Write ASCII and proceed
C1	Read binary and proceed
C2	Write binary and proceed

Appendix E

HEXADECIMAL CONVERSION TABLES

ASCII CODE CONVERSION

HEXADECIMAL

LEAST SIGNIFICANT CHARACTER	MOST SIGNIFICANT CHARACTER								
	—	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

Example

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

DECIMAL-HEXADECIMAL-BINARY EQUIVALENTS 0—255

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

HEXADECIMAL ADDITION

	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

Example

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

HEXADECIMAL MULTIPLICATION

	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

Example

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

Appendix F

ASSEMBLER ERROR CODES

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

Indicates that a user entered WARNING message has assembled. Refer to WARNING directive explanation in Section 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 assignment of a label or EQU symbol value differs between passes.

***** ERROR: 004 Illegal EQU of GLOBALS

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 directive 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 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 secod 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 defintion, 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 that an END directive is present in 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 a REPEAT block, or when IF and REPEAT blocks overlap each other.

***** 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 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

***** 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. Address values do not cause the error to be displayed.

***** ERROR: 036 Invalid Character Follows Label

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

***** ERROR: 037 Extra Operands 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 parenthesis ")". 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 prevents the operation from being 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 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 Appendices B and C for required syntax for Assembler directives and 8080A/8085A instructions.

***** 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 are used in place of commas to delimit options, or when an invalid listing control option is entered.

***** ERROR: 043 No Label on EQU or SET

Indicates that a symbol is either missing from or invalid for the label field of an EQU or SET 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 operation (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 parenthesis "(", 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 Sections 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 Operation

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

***** ERROR: 054 Negative RESERVE Length

Indicates that a negative-value 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 or 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 Section 4.

***** ERROR: 061 GLOBAL Declaration Phase Error

Indicates that a symbol was not declared global 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 8002A μProcessor Lab Assembler recognizes only hexadecimal (H), octal (Q) or (O), and binary (B) 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 that an 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 parenthesis "(", 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 SET Strings, Section 2.

***** ERROR: 076 Negative SEG Operand

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

***** ERROR: 077 SEG Starting Operand is Zero

Indicates a zero in the starting position of the SEG operand. See SEG, Section 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 space directive's operand value 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 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 discontinues processing the program.

***** ERROR: 088 Invalid Transfer Label

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

***** ERROR: 091 Unable to Assign INCLUDE File

Indicates that TEKDOS 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 TEKDOS could not assign the file.

The following error messages apply only to the 8080A/8085A:

***** ERROR: 254 Register Expression is Not Scalar

Indicates that an address expression is used where a register expression is required.

***** ERROR: 253 Invalid Register Pair

Indicates that the specified register either is greater than the number 6, represents an odd value, or is an invalid register pair for the specified instruction.

***** ERROR: 252 Register Expression Greater Than 7

Indicates that the value assigned to the specified register exceeds the number 7.

***** ERROR: 251 Missing or Invalid Operand

Indicates a missing operand or a syntax error in the current operand.

Appendix G

RESERVED WORDS

8080A/8085A microprocessor instruction names, register symbol names, and Tektronix Assembler Directive names should not be used as symbolic labels. The following names are reserved for these special uses:

8080A/8085A INSTRUCTIONS

ACI	CMC	DAD	JM	LHLD	PUSH	RP	SPHL
ADC	CMP	DCR	JMP	LXI	RAL	RPE	STA
ADD	CNC	DCX	JNC	MOV	RAR	RPO	STAX
ADI	CNZ	DI	JNZ	MVI	RC	RRC	STC
ANA	CP	EI	JP	NOP	RET	RST	SUB
ANI	CPE	HLT	JPE	ORA	RIM	RZ	SUI
CALL	CPI	IN	JPO	ORI	RLC	SBB	XCHG
CC	CPO	INR	JZ	OUT	RM	SBI	XRA
CM	CZ	INX	LDA	PCHL	RNC	SHLD	XRI
CMA	DAA	JC	LDAX	POP	RNZ	SIM	XTHL

8080A/8085A REGISTER SYMBOL

A	B	C	D	E
H	L	M	PSW	SP

TEKTRONIX ASSEMBLER DIRECTIVES, OPTIONS and OPERATORS

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

FUTURE TEKTRONIX ASSEMBLER DIRECTIVES (TENTATIVE)

XREF

Appendix H

OBJECT CODE OF INSTRUCTIONS

HEX	OP	BYTES 2-3	HEX	OP	BYTES 2-3	HEX	OP	BYTES 2-3
00	NOP		37	STC		6B	MOV L,E	
01	LXI B	data-16	39	DAD SP		6C	MOV L,H	
02	STAX B		3A	LDA	address	6D	MOV L,L	
03	INX B		3B	DCX SP		6E	MOV L,M	
04	INR B		3C	INR A		6F	MOV L,A	
05	DCR B		3D	DCR A		70	MOV M,B	
06	MVI B	data-8	3E	MVI A	data-8	71	MOV M,C	
07	RLC		3F	CMC		72	MOV M,D	
09	DAD B		40	MOV B,B		73	MOV M,E	
0A	LDAX B		41	MOV B,C		74	MOV M,H	
0B	DCX B		42	MOV B,D		75	MOV M.L	
0C	INR C		43	MOV B,E		76	HLT	
0D	DCR C		44	MOV B,H		77	MOV M,A	
0E	MVIC	data-8	45	MOV B,L		78	MOV A,B	
0F	RRC		46	MOV B,M		79	MOV A,C	
11	LXI D	data-16	47	MOV B,A		7A	MOV A,D	
12	STAX D		48	MOV C,B		7B	MOV A,E	
13	INX D		49	MOV C,C		7C	MOV A,H	
14	INR D		4A	MOV C,D		7D	MOV A,L	
15	DCR D		4B	MOV C,E		7E	MOV A,M	
16	MVI D	data-8	4C	MOV C,H		7F	MOV A,A	
17	RAL		4D	MOV C,L		80	ADD B	
19	DAD D		4E	MOV C,M		81	ADD C	
1A	LDAX D		4F	MOV C,A		82	ADD D	
1B	DCX D		50	MOV D,B		83	ADD E	
1C	INR E		51	MOV D,C		84	ADD H	
1D	DCR E		52	MOV D,D		85	ADD L	
1E	MVI E	data-8	53	MOV D,E		86	ADD M	
1F	RAR		54	MOV D,H		87	ADD A	
20	RIM		55	MOV D,L		88	ADC B	
21	LXI H	data-16	56	MOV D,M		89	ADC C	
22	SHLD	address	57	MOV D,A		8A	ADC D	
23	INX H		58	MOV E,B		8B	ADC E	
24	INR H		59	MOV E,C		8C	ADC H	
25	DCR H		5A	MOV E,D		8D	ADC L	
26	MVI H	data-8	5B	MOV E,E		8E	ADC M	
27	DAA		5C	MOV E,H		8F	ADC A	
29	DAD H		5D	MOV E,L		90	SUB B	
2A	LHLD	address	5E	MOV E,M		91	SUB C	
2B	DCX H		5F	MOV E,A		92	SUB D	
2C	INR L		60	MOV H,B		93	SUB E	
2D	DCR L		61	MOV H,C		94	SUB H	
2E	MVI L	data-8	62	MOV H,D		95	SUB L	
2F	CMA		63	MOV H,E		96	SUB M	
30	SIM		64	MOV H,H		97	SUB A	
31	LXI SP	data-16	65	MOV H,L		98	SBB B	
32	STA	address	66	MOV H,M		99	SBB C	
33	INX SP		67	MOV H,A		9A	SBB D	
34	INR M		68	MOV L,B		9B	SBB E	
35	DCR M		69	MOV L,C		9C	SBB H	
36	MVI M	data-8	6A	MOV L,D		9D	SBB L	
9E	SBB M							

HEX	OP	BYTES 2-3
9E	SBB M	
9F	SBB A	
A0	ANA B	
A1	ANA C	
A2	ANA D	
A3	ANA E	
A4	ANA H	
A5	ANA L	
A6	ANA M	
A7	ANA A	
A8	XRA B	
A9	XRA C	
AA	XRA D	
AB	XRA E	
AC	XRA H	
AD	XRA L	
AE	XRA M	
AF	XRA A	
B0	ORA B	
B1	ORA C	
B2	ORA D	
B3	ORA E	
B4	ORA H	
B5	ORA L	
B6	ORA M	
B7	ORA A	
B8	CMP B	
B9	CMP C	
BA	CMP D	
BB	CMP E	
BC	CMP H	
BD	CMP L	
BE	CMP M	
BF	CMP A	
C0	RNZ	
C1	POP B	
C2	JNZ	address
C3	JMP	address
C4	CNZ	address
C5	PUSH B	
C6	ADI	data-8
C7	RST 0	
C8	RZ	
C9	RET	address
CA	JZ	
CC	CZ	address
CD	CALL	address
CE	ACI	data-8
CF	RST 1	
D0	RNC	
D1	POP D	
D2	JNC	address
D3	OUT	data-8
D4	CNC	address
D5	PUSH D	
D6	SUI	data-8
D7	RST 2	
D8	RC	
DA	JC	address
DB	IN	data-8
DC	CC	address
DE	SBI	data-8
DF	RST 3	
E0	RPO	
E1	POP H	
E2	JPO	address
E3	XTHL	
E4	COP	address
E5	PUSH H	
E6	ANI	data-8
E7	RST 4	
E8	RPE	
E9	PCHL	
EA	JPE	address
EB	XCHG	
EC	CPE	address
EE	XRI	data-8
EF	RST 5	
F0	RP	
F1	POP PSW	
F2	JP	address
F3	DI	
F4	CP	
F5	PUSH PSW	
F6	ORI	data-8
F7	RST 6	
F8	RM	
F9	SPHL	
FA	JM	address
FB	EI	
FC	CM	address
FE	CPI	data-8
FF	RST 7	

Mnemonic	Hex	Mnemonic	Hex	Mnemonic	Hex
JNZ	C2	CNZ	C4	RNZ	C0
JZ	CA	CZ	CC	RZ	C8
JNC	D2	CNC	D4	RNC	D0
JC	DA	CC	DC	RC	D8
JPO	E2	CPO	E4	RPO	E0
JPE	EA	CPE	EC	RPE	E8
JP	F2	CP	F4	RP	F0
JM	FA	CM	FC	RM	F8