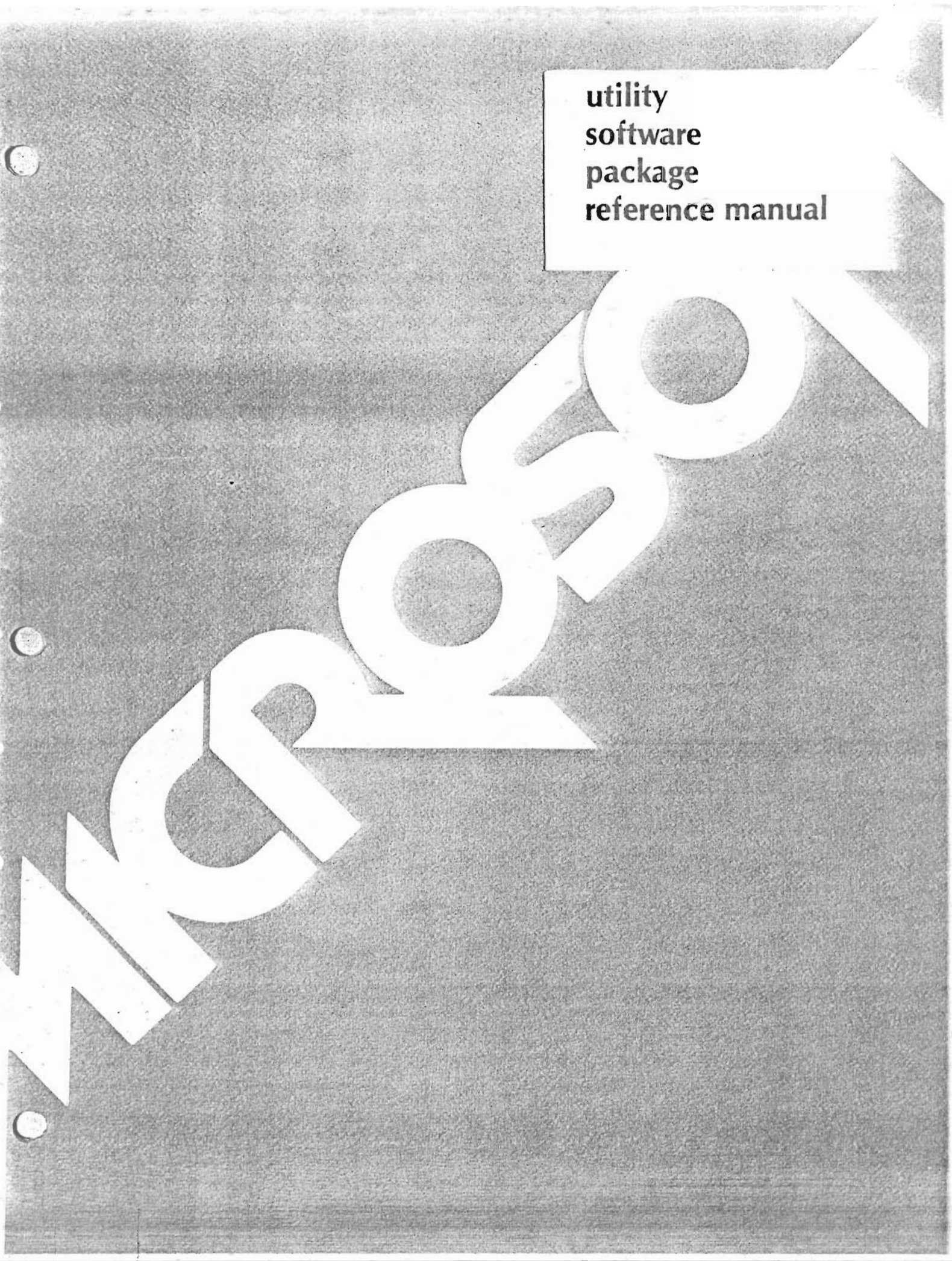
utility software package reference manual



Contents

Contents

CHAPTER 1

INTRODUCTION

Welcome to the world of Utility Software Package programming. During the course of this manual, we will learn what the Utility Software Package is, why you use it, and how to use it.

1.1 CONTENTS OF THE UTILITY SOFTWARE PACKAGE

One diskette with the following files:

M80.COM - MACRO-80 Macro Assembler program
L80.COM - LINK-80 Linking Loader program
CREF80.COM - Cross-Reference Facility
LIB.COM - Library Manager program
(CP/M versions only)

One Manual

The Utility Software Package Reference Manual

IMPORTANT

> Always make backup copies of your diskettes before using them.

1.2 SYSTEM REQUIREMENTS

MACRO-80 requires about 19K of memory, plus about 4K for buffers. LINK-80 requires about 14K of memory. CREF-80 requires about 6K of memory. LIB-80 requires about 5K of memory. The operating system usually requires about 6K bytes of memory. So a minimum system requirement for the Utility Software Package is 29K bytes of memory. While it is possible to run Utility Software Package programs with only one disk drive, we recommend strongly that you have two disk drives available.

1.3 WHOM IS THE UTILITY SOFTWARE PACKAGE FOR?

The Utility Software Package is a complete assembly language development system with powerful features that support advanced assembly language programming skills. This manual describes the Utility Software Package thoroughly, but the descriptions assume that the reader understands assembly language programming and has experience with an assembler.

If you have never programmed in assembly language, we suggest that you gain some experience on a simpler assembler.

Books on Assembly Language Programming

We can also recommend the following books for basic instruction in assembly language programming:

Leventhal, Lance A. 8080A/8085 Assembly Language Programming. Berkeley: Osborne/McGraw-Hill, 1978.

Leventhal, Lance A. Z80 Assembly Language Programming. Berkeley: Osborne/McGraw-Hill, 1979.

Zaks, Rodnay. Programming the Z80. Second edition. Berkeley: Sybex, 1980.

1.4 A WORD ABOUT THIS MANUAL

Organization

In front of each chapter is a contents page that expands the entries on the contents page at the beginning of the manual. Chapter 1 gives introductory, background, and overview information about the Utility Software Package. Chapters 2-8 describe the use and operation of the Utility Software Package programs. The manual concludes with several appendices which contain some helpful reference information.

Syntax Notation

The following notation is used throughout this manual in descriptions of command and statement syntax:

[ ] Square brackets indicate that the enclosed entry is optional.

< > Angle brackets indicate user entered data. When the angle brackets enclose lower case text, the user must type in an entry defined by the text; for example, <filename>. When the angle brackets enclose upper case text, the user must press the key named by the text; for example, <RETURN>.

{ } Braces indicate that the user has a choice between two or more entries. At least one of the entries enclosed in braces must be chosen unless the entries are also enclosed in square brackets.

... Ellipses indicate that an entry may be repeated as many times as needed or desired.

CAPS Capital letters indicate portions of statements or commands that must be entered, exactly as shown.

All other punctuation, such as commas, colons, slash marks, and equal signs, must be entered exactly as shown.

1.5 OVERVIEW

The Utility Software Package is an assembly language programming system that parallels the design and programming power of assemblers and related software on big computers. Consequently, the design and use of the Utility Software Package involves traits and methods that may be new to you. As explained earlier, we assume that you have some experience in assembly language programming. Your knowledge of when and why to use particular operation codes and pseudo-operations is the base on which you can build your knowledge of the Utility Software Package.

One word of caution: some terms used in this manual may be familiar to you from other sources. Be sure to notice especially how familiar terms are used in the Utility Software Package so that you are not confused or misled.

The Utility Software Package programming relies on two important software programs -- an assembler and a linking loader. To develop an assembly language program that runs on your computer, you must use both the assembler and the linking loader. The whole process is diagrammed on the facing page. The numbers on the diagram correspond to the numbers in the explanations below.

1. You create an assembly language source program using some editor.

2. You assemble your source program using the MACRO-80 macro assembler. The result is a file that contains intermediate object code. This intermediate code is closer to machine code than your source code, but cannot be executed.

3. You link and load separately assembled file(s) into a single program file using the LINK-80 linking loader. LINK-80 converts the file(s) of intermediate code into a single file of true machine code which can be executed from the operating system.

These are only the basics of the whole process. This two step process of converting a source file to an executable program allows you to manipulate your programs to save you time and to extend your programs' usefulness in the following ways:

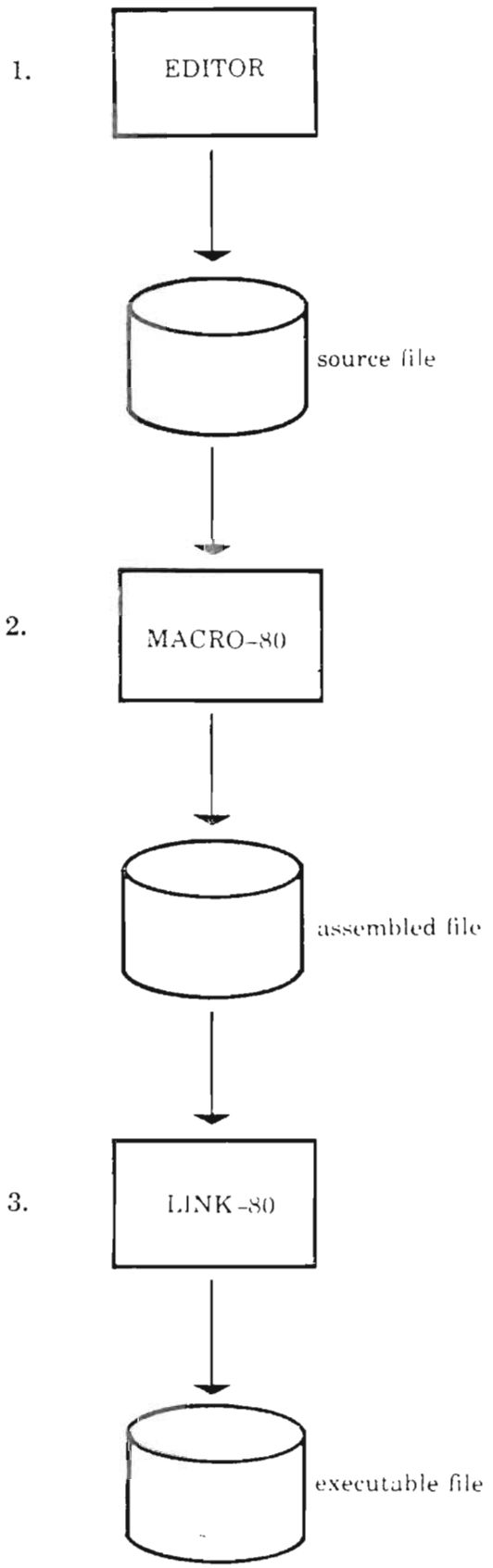


Figure 1.1: Developing Assembly Language Programs

First, you can break your program in convenient parts called modules. You can manipulate these modules at will. You can assemble the modules individually. You fix only those that do not work right and reassemble them. This saves you time.

Second, you can manipulate the placement of modules in memory, subject to certain restrictions; or allow LINK-80 to place modules for you. (This trait is described below under the fourth trait.)

Third, you can use assembled modules in other programs or in variations of the original program because there is no permanent connection among the modules. This saves you recoding time if a part of a program performs some useful, often-repeated task.

Whenever you want to combine assembled modules into an executable program, you use the LINK-80 linking loader. If you simply tell LINK-80 the modules you want combined, it loads them end-to-end in memory. But you have an additional choice. You can set up a direct connection between a statement in one module and a statement inside another module. This direct connection (or "link") means that a value (usually a program address) in one module can be used in another module exactly at the point required.

LINK-80 creates the links between modules. You give LINK-80 the signals it needs to create these links. The signals are called symbols, specifically EXTERNAL symbols and PUBLIC symbols. An EXTERNAL symbol signals LINK-80 that you want it to link a value from another module into this point in the program. The value to be linked-in is defined by a PUBLIC symbol, which is a signal that directs LINK-80 to the correct module and statement line. LINK-80 then links the PUBLIC symbol's value to the EXTERNAL symbol, then continues loading the module with the EXTERNAL symbol. The diagram below suggests this process.

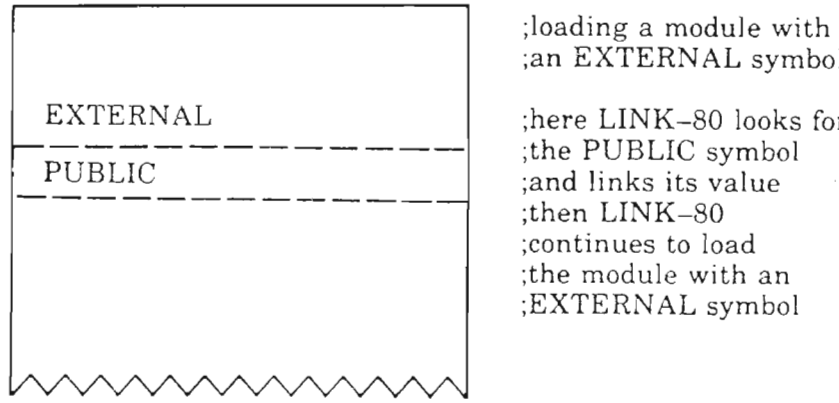


Figure 1.2: PUBLIC symbol linked into module at EXTERNAL

Fourth, modules can be assembled into different modes, even within a single module. The four modes are Absolute, Data-relative, Code-relative, and COMMON-relative. The absolute mode is similar the code produced by most small system assemblers. The code is assembled at fixed addresses in memory. The other three modes are very different and are the reason you can place modules anywhere in memory. Each of the three relative modes assembles to a separate segment. The addresses within each segment are relative addresses. This means the first instruction byte of a segment is given a relative address of 0, the second byte is given relative address 1, and so on. When LINK-80 loads the module, it changes the relative addresses in the segments to fixed addresses in memory. The relative addresses are offsets from some fixed address that LINK-80 uses. For the first module loaded, this address is 103H under the CP/M operating system. Thus, relative addresses in the first module are offsets from 103H. The second module is loaded at the end of the first, and the relative addresses are offsets from the last address in the first module. Subsequent modules are loaded (and offset) similarly. You can change the default start address for the first module at link time. Then, the relative addresses become offsets from the fixed address you specify.

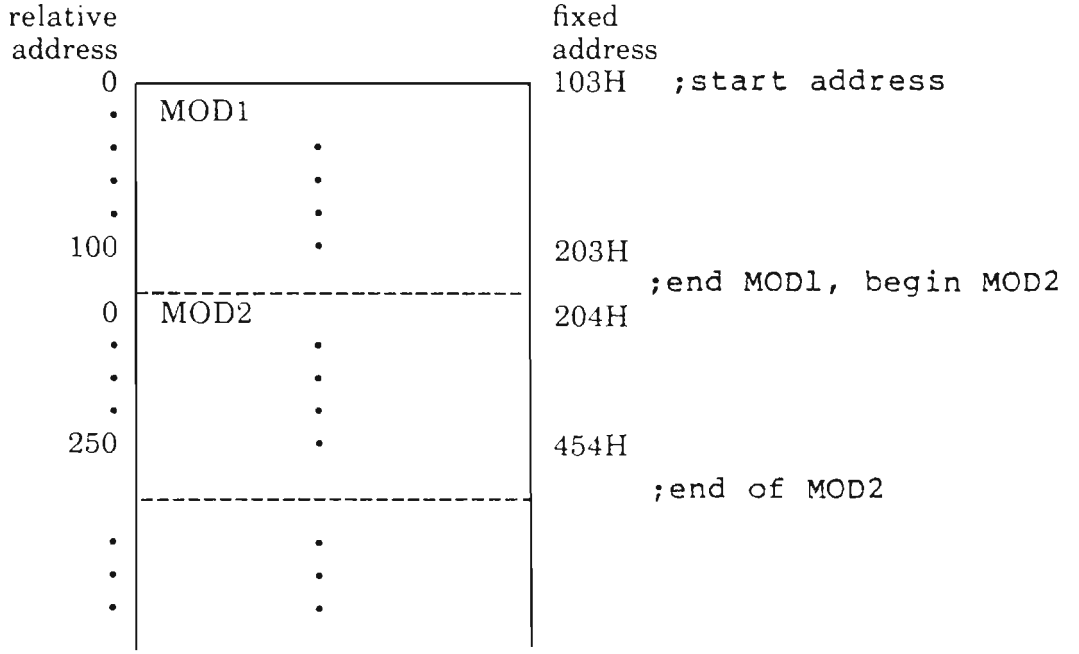
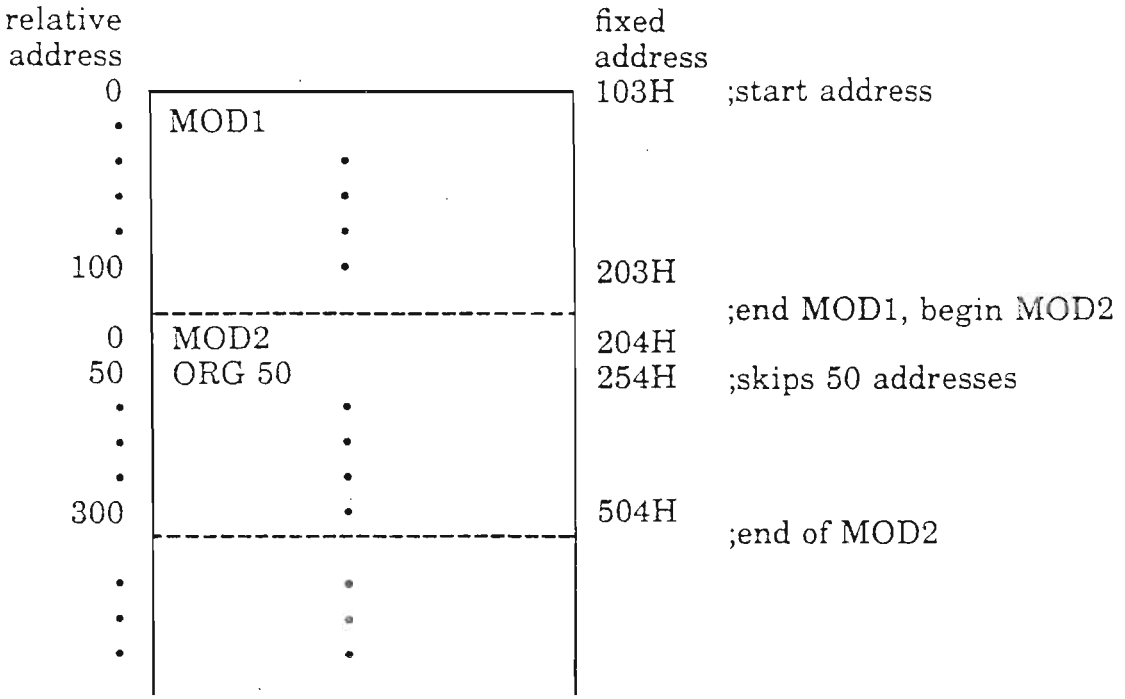


Figure 1.3: Loading Changes Relative Addresses to Fixed

One effect of this relative addressing method is that ORG statements become very different creatures. For the relative segments, the ORG statement specifies an offset rather than a fixed address (as most assemblers do -- ORG specifies a fixed address in the absolute segment). Thus, a relative segment with an ORG statement would skip over a specified number of addresses before beginning to load the rest of the code in that segment.



You should read carefully the description of ORG found in Chapter 4.

The ability to manipulate the placement of modules in memory, with some restrictions (see Chapter 6), derives from the assembler giving relative addresses instead of absolute addresses. This ability to manipulate module placement in memory is called relocatability; the modules are relocatable; the intermediate code produced by the assembler for the three relative segments is called relocatable code. That is why assembled modules are given the filename extension .REL, and these assembled files are called REL files.

Each mode serves a different purpose. The absolute mode contains code you want placed in specific memory addresses. Each relative mode is loaded into memory as a separate segment. The data-relative segment contains data items and any code that may change often and should only be placed in RAM. The code-relative segment contains code that will not change and therefore is suitable for ROM and PROM. The COMMON-relative segment contains data items that can be shared by more than one module.

Source statements in these modes take on the traits of their mode. The symbols and expressions in statements are evaluated by the assembler according to the mode in which they are found and the type of data and other entries that define the symbol or make up the parts of an expression. The mode traits attributed to a symbol or expression are called, appropriately, its Mode; that is, a symbol or expression is absolute, data-relative, code-relative, or COMMON-relative. This concept of mode is important because it is the source of both flexibility and complexity. If all

source statements are assembled in absolute mode, symbols and expressions always have absolute values, and using absolute symbols and expressions is not complex. The problem with absolute mode is that relocatability is possible only through the most complex and time consuming of techniques. Absolute mode effectively reduces your ability to reuse code in a new program.

The relative modes (data, code, and COMMON) are the basis of relocatability and, therefore, of the flexibility to manipulate modules. The complexity is that relative symbols and relative expressions are much more difficult to evaluate. In fact, the assembler must pass through the source statements twice to assemble a module. During the first pass, the assembler evaluates the statements and expands macro call statemnts, calculates the amount of code it will generate, and builds a symbol table where all symbols and macros are assigned values. During the second pass, the assembler fills in the symbol and expression values from the symbol table, expands macro call statements, and emits the intermediate code into a REL file.

When the REL files are given to LINK-80, the segments are linked together and loaded into fixed memory addresses. The relative addresses are converted to absolute addresses. The fixed addresses are assigned to the relative segments in the order: COMMON-relative and data-relative, then code-relative. The relative segments are loaded relative to default address 103H under CP/M. (The addresses 100H-102H are used for a jump to the start address of the first program instruction, which is normally the first address following the COMMON and data area.)

When LINK-80 is finished linking modules together and assigning addresses, the result can be saved in a file that is executable from the operating system. Executing the program is then as simple as entering an operating system command, so these linked and loaded files are called command files.

This short overview should give you a general idea of the workings and processes of the Utility Software Package. Short descriptions of all the Utility Software Package programs are given in the next chapter. Detailed descriptions are given in the rest of this manual. Therefore, the information contained in this overview will be repeated in fuller detail elsewhere in this manual.

As an aid to the description in the next chapter and the rest of this manual, the next page contains an expanded version of the diagram at the beginning of this overview. The expanded diagram shows the relationships among all the programs in the Utility Software Package.

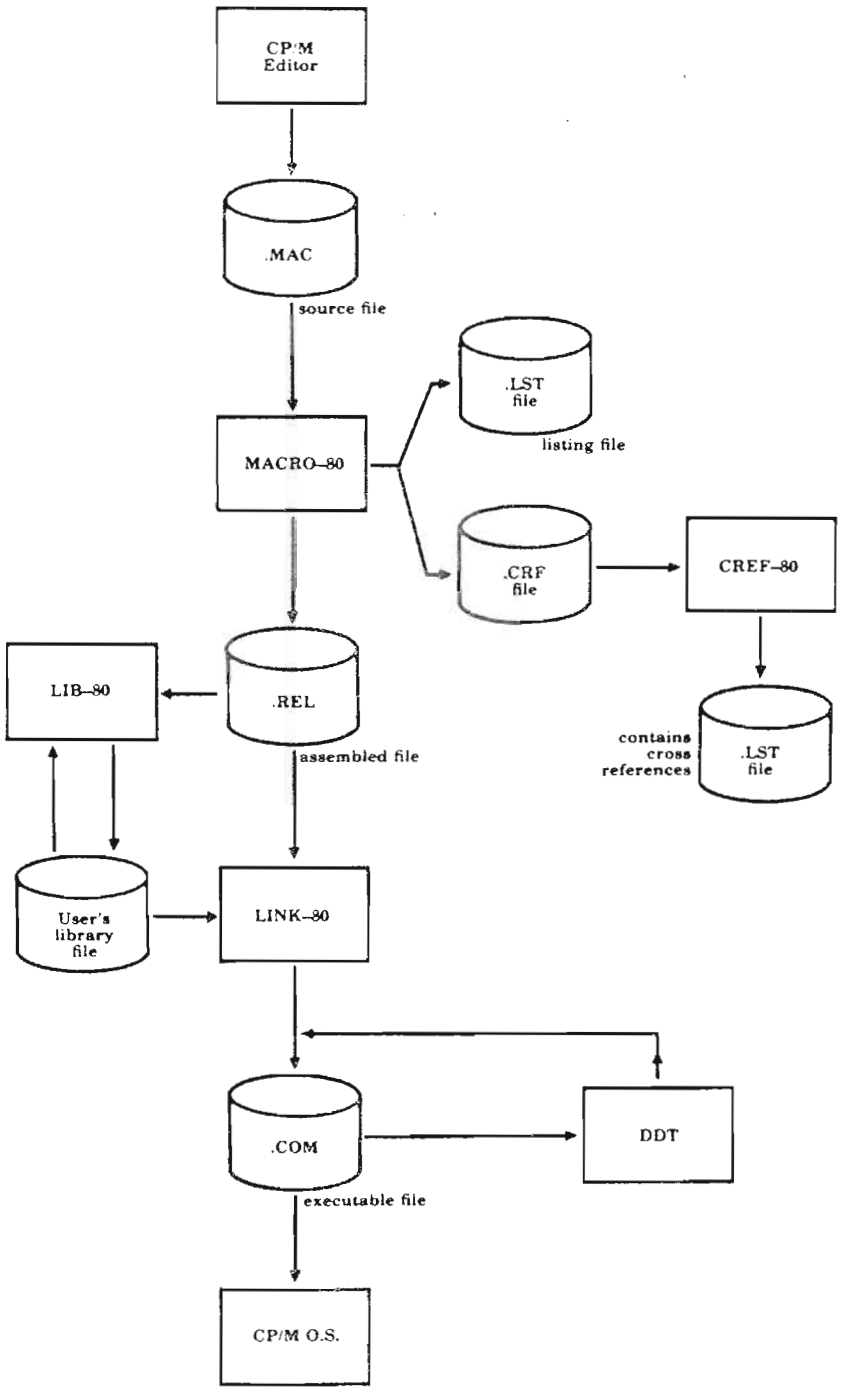


Figure 1.5: Relationships among programs

Contents

CHAPTER 2

FEATURES OF THE UTILITY SOFTWARE PACKAGE

The Utility Software Package is an Assembly Language Development System that assembles relocatable code from two assembly languages, supports a macro facility and conditional assembly, and provides several utility programs that enhance program development.

WHAT IS AN UTILITY SOFTWARE PACKAGE?

An Utility software package is more than an assembler. An Utility Software Package is a series of related utility programming tools:

- for assembling an assembly language source file,
- for linking several assembled modules into one program,
- for creating library files of subroutines (also assembled modules),
- for creating cross-reference listings of program symbols,
- for testing and debugging binary (machine executable) program files,

Microsoft's Utility Software Package provides versions of these tools that make the Utility Software Package extremely powerful and useful as a program development system. Each tool in the Utility Software Package is described in detail in its own chapter.

2.1 TWO ASSEMBLY LANGUAGES

The assembler in your Utility Software Package supports two assembly languages. Microsoft's MACRO-80 macro assembler supports both 8080 and Z80 mnemonics.

2.2 RELOCATABILITY

MACRO-80 can produce modules of relocatable code. Also, like many assemblers, the MACRO-80 assembler can produce absolute code. The key advantage of relocatability is that programs can be assembled in modules. Then, within certain restrictions described in Chapter 6, the modules can then be located almost anywhere in memory.

Relocatable modules also offer the advantages of easier coding and faster testing, debugging, and modifying. In addition, it is possible to specify segments of assembled code that will later be loaded into RAM or into ROM/PROM.

Relocatability will be discussed further under Section 3.2, Symbols.

2.3 MACRO FACILITY

The MACRO-80 assembler supports a complete, Intel standard macro facility. The macro facility allows a programmer to write blocks of code for a set of instructions used frequently. The need for recoding these instructions is eliminated.

The programmer gives this block of code a name, called a macro. The instructions are the macro definition. Each time the set of instructions is needed, instead of recoding the set of instructions, the programmer simply "calls" the macro. MACRO-80 expands the macro call by assembling the block of instructions into the program automatically. The macro call also passes parameters to the assembler for use during macro expansion. The use of macros reduces the size of a source module because the macro definitions are stored in disk files and come into the module only when needed during assembly.

Macros can be nested, that is, a macro can be called from inside another macro. Nesting of macros is limited only by memory.

2.4 CONDITIONAL ASSEMBLY

MACRO-80 also supports conditional assembly. The programmer can determine a condition under which portions of the program are either assembled or not assembled. Conditional assembly capability is enhanced by a complete set of conditional pseudo operations that include testing of assembly pass, symbol definition, and parameters to macros. Conditionals may be nested up to 255 levels.

2.5 UTILITY PROGRAMS

Three utility programs provide the additional support needed to develop powerful and useful assembly language programs: LINK-80 Linking Loader, LIB-80 Library Manager, and CREF-80 Cross Reference Facility.

LINK-80 Linking Loader

The Microsoft LINK-80 Linking Loader is used to convert the assembled module (.REL file) into an executable module (.COM file). The .REL file is not an executable file.

LINK-80 can also be used to:

load, link, and run one or more modules

load relocatable programs at user-specified locations

load program areas and data areas into separate memory locations

While performing these tasks, LINK-80 resolves external references between modules (that is, any program that calls an external value, something defined in a different program or module, will have the outside references filled at link time by LINK-80), and saves the executable object (.COM) file on disk, so it can be run from the operating system.

These load capabilities mean that the assembled program may be linked with the user's library to add routines to one of the high-level langauge runtime libraries. Assembled programs can be linked to high-level language programs -- COBOL-80 and FORTRAN-80, for example -- as well as to MACRO-80 programs.

CREF-80 Cross Reference Facility

The CREF-80 Cross Reference Facility processes a cross reference file generated by MACRO-80. The result is a cross reference listing that can aid in the debugging of your program.

LIB-80 Library Manager (CP/M versions only)

LIB-80 is designed as a runtime library manager for CP/M versions of the Utility Software Package. LIB-80 may also be used to create your own library of assembly language subroutines.

LIB-80 creates runtime libraries from assembly language programs that are subroutines to COBOL, FORTRAN, and other assembly language programs. The programs collected by LIB-80 may be special modules created by the programmer or modules from an existing library. With LIB-80, you can create specialized runtime libraries for whatever execution requirements you design.

Contents

CHAPTER 3

PROGRAMMING WITH THE UTILITY SOFTWARE PACKAGE

This chapter describes what the user needs to know to create MACRO-80 macro assembler source files. Source files are created using a text editor, such as CP/M ED. The Utility Software Package does not include a text editor program.

Source files are assembled using the procedures described in Chapter 4.

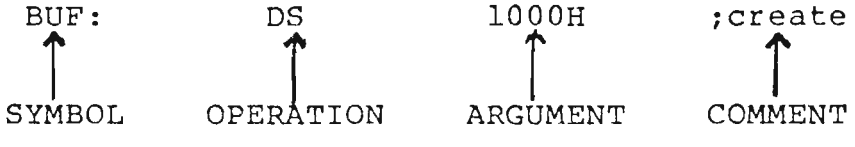
3.1 SOURCE FILE ORGANIZATION

File Organization

A MACRO-80 macro assembler source file is a series of lines written in assembly language. The last line of the file must be an END statement. Matching statements (such as IF...ENDIF) must be entered in the proper sequence. Otherwise, lines may appear in any order the programmer designs.

Statement Line Format

Source files input to the MACRO-80 macro assembler consist of statement lines divided into parts or "fields."



SYMBOL field contains one of the three types of symbol (LABEL, PUBLIC, and EXTERNAL), followed by a colon unless it is part of a SET, EQU, or MACRO statement.

OPERATION field contains an OPCODE, a PSEUDO-OP, a MACRO name, or an expression.

ARGUMENT field contains expressions (specific values, variables, register names, operands and operators).

;COMMENT field contains comment text always preceded by a semicolon.

All fields are optional. You may enter a completely blank line.

Statement lines may begin in any column. Multiple blanks or tabs may be inserted between fields to improve readability, but at least one space or tab is required between each field.

Comments

A MACRO-80 macro assembler source line is basically an Operation and its Argument. Therefore, the MACRO-80 macro assembler requires that a COMMENT always begin with a semicolon. A COMMENT ends with a carriage return.

For long comments, you may want to use the .COMMENT pseudo-op to avoid entering a semicolon for every line. See the File Related Pseudo-ops section of Chapter 4 for the description of .COMMENT.

3.2 SYMBOLS

Symbols are simply names for particular functions or values. Symbol names are created and defined by the programmer.

Symbols in the Utility Software Package belong to one of three types, according to their function. The three types are LABEL, PUBLIC, and EXTERNAL. All three types of symbols have a MODE attribute that corresponds to the segment of memory the symbol represents. Refer to the section on modes following the description of symbol types.

All three types of symbols have the following characteristics:

1. Symbols may be any length, but the number of significant characters passed to the linker varies with the type of symbol:

   a. for LABELs, only the first sixteen characters are significant.

   b. for PUBLIC and EXTERNAL symbols, only the first six characters are passed to the linker.

   Additional characters are truncated internally.

2. A legal symbol name may contain the characters:

   A-Z 0-9 $. ? @ _

3. A symbol may not start with a digit or an underline

4. When a symbol is read, lower case is translated into upper case, so you may enter the name using either case or both.

LABEL:

A LABEL: is a reference point for statements inside the program module where the label appears. A LABEL: sets the value of the symbol LABEL to the address of the data that follows. For example, in the statement:

```
BUF:    DS    1000H
```

BUF: equals the first address of the 1000H byte reserved space.

Once a label is defined, the label can be used as an entry in the ARGUMENT field. A statement with a label in its argument loops to the statement line with that label in its SYMBOL field, which is where the label is defined. The label's definition replaces the label used in an ARGUMENT field. For example,

```
STA    BUF
```

sends the value in the accumulator to the area in memory represented by the label BUF.

A LABEL may be any legal symbol name, up to 16 characters long.

If you want to define a LABEL, it must be the first item in the statement line. 8080 and Z80 labels must be followed immediately by a single colon (no space), unless the LABEL is part of a SET or EQU statement. (If two colons are entered, the "label" becomes a PUBLIC symbol. See PUBLIC Symbols below.)

PUBLIC

A PUBLIC symbol is defined much like a LABEL. The difference is that a PUBLIC symbol is available as a reference point for statements in other program modules, too.

A symbol is declared PUBLIC by:

two colons (::) following the name. For example,

```
FOO::       RET
```

one of the pseduo-ops PUBLIC, ENTRY, or GLOBAL. For example,

```
PUBLIC      FOO
```

See the Data Definition and Symbol Definition Pseudo-ops section in Chapter 4 for descriptions of how to use these pseudo-ops.

The result of both methods of declaration is the same. Therefore,

```
FOO::       RET
```

is equivalent to

```
PUBLIC      FOO
FOO:        RET
```

EXTERNAL

An EXTERNAL symbol is defined outside the program module where it appears. An EXTERNAL symbol is defined as a PUBLIC symbol in another, separate program module. At link time (when the LINK-80 Linking Loader is used), the EXTERNAL symbol is given the value of the PUBLIC symbol in the other program module. For example:

MOD1

```
FOO::     DB     7     ;PUBLIC FOO = 7
```

MOD2

```
          BYTE EXT     FOO     ;EXTERNAL FOO
```

At link time, LINK-80 goes to the address of PUBLIC FOO and uses the value there (7) for EXTERNAL FOO.

A symbol is declared EXTERNAL by:

1. two pound signs (##) following a reference to a symbol name. For example:

   ```
   CALL    FOO##
   ```

   declares FOO as a two-byte symbol defined in another program module.

2. one of the pseudo-ops EXT, EXTRN, or EXTERNAL for two-byte values. For example:

   ```
   EXT    FOO
   ```

   declares FOO as a two-byte value defined in another program module.

3. one of the pseudo-ops BYTE EXT, BYTE EXTERN, or BYTE EXTERNAL for one-byte values. For example:

   ```
   BYTE EXT     FOO
   ```

   declares FOO as a one-byte value defined in another program module.

   See the Symbol Definition Pseudo-ops section in Chapter 4 for descriptions of how to use these pseudo-ops.

As for PUBLIC symbols, the result of both methods of declaration is the same. Therefore,

```
CALL    FOO##
```

is equivalent to

```
EXT     FOO
CALL    FOO
```

MODES

A symbol is referenced by entering its name in the ARGUMENT field of a statement line. When a symbol is referenced, the value of the symbol (derived from the instruction which defines the symbol) is substituted for the symbol name and used in the operation.

The value of a symbol is evaluated according to its program counter (PC) mode. The PC mode determines whether a section of a program will be loaded into memory at addresses predetermined by the programmer (absolute mode), or at relative addresses that change depending on the size and number of programs (code relative mode) and amount of data (data relative mode), or at addresses shared with another program module (COMMON mode). The default mode is Code Relative.

Absolute Mode: Absolute mode assembles non-relocatable code. A programmer selects Absolute mode when a block of program code is to be loaded each time into specific addresses, regardless of what else is loaded concurrently.

Data Relative Mode: Data Relative mode assembles code for a section of a program that may change and therefore must be loaded into RAM. This applies to program data areas especially. Symbols in Data Relative Mode are relocatable.

Code Relative Mode: Code (program) Relative mode assembles code for sections of programs that will not be changed and therefore can be loaded into ROM/PROM. Symbols in Code Relative Mode are relocatable.

COMMON Mode: COMMON mode assembles code that is loaded into a defined common data area. This allows program modules to share a block of memory and common values.

To change mode, use a PC mode pseudo-op in a statement line. The PC mode pseudo-ops are:

```
ASEG    Absolute mode
DSEG    Data Relative mode
CSEG    Code Relative mode--default mode
COMMON  COMMON mode
```

These pseudo-ops are described in detail in the PC Mode Pseudo-ops section of Chapter 4.

This PC mode capability in the MACRO-80 macro assembler allows a programmer to develop assembly language programs that can be relocated. Many assembly language programmers may have learned always to set an Origin statement at the beginning of every module, subroutine, or main assembly language program. Under MACRO-80 this mode of addressing is called Absolute mode because hard (or actual addresses) are specified beginning, especially, with the Origin statement.

MACRO-80 has two other, "relative" modes of addressing available, called Code (Program) relative and Data relative. Segments of code written in these two modes are relocatable. Relocatable means the program module can be loaded starting at any address in available memory, using the /P and /D switches (special commands) in LINK-80.

3.3 OPCODES AND PSEUDO-OPS

Opcodes are the mnemonic names for the machine instructions. Pseudo-ops are directions to the assembler, not the microprocessor.

MACRO-80 supports two instruction sets: 8080 and Z80. A list of the opcodes with brief summaries of their functions is included as Appendix F. To program with the opcodes of the different languages, the user must first enter the pseudo-op which tells the assembler which language is being coded. Refer to the Language Set Selection Pseudo-ops section of Chapter 4 for details.

MACRO-80 also supports a large variety of pseudo-ops that direct the assembler to perform many different functions. The pseudo-ops are described extensively in Chapter 4 and are summarized in Appendix E.

Opcodes and pseudo-ops are (usually) entered in the OPERATION field of a statement line. (A program statement line usually has an entry in the operation field, unless the line is a Comment line only. The Operation field will be the first field filled if no label is entered.) An Operation may be any 8080 or Z80 mnemonic; or a MACRO-80 macro assembler pseudo-op, macro call, or expression.

The OPERATION field entries are evaluated in the following order:

1. Macro call

2. Opcode/Pseudo-op

3. Expressions

MACRO-80 compares the entry in the OPERATION filed to an internal list of macro names. If the entry is found, the macro is expanded. If the entry is not a macro, MACRO-80 tries to evaluate the entry as an opcode. If the entry is not an opcode, MACRO-80 tries to evaluate the entry as a pseudo-op. If the entry is not a pseudo-op, MACRO-80 evaluates the entry as an expression. If an expression is entered as a statement line without an opcode, pseudo-op, or macro name in front of it, the MACRO-80 macro assembler does not return an error. Rather, the assembler assumes that a define byte pseudo-op belongs in front of the expression and assembles the line.

Because of the order of evaluation, a macro name that is the same as an opcode prevents you from using the opcode again, except as a macro call. For example, if you give a block of macro code the name ADD in your program, you cannot use ADD as an opcode in that program.

3.4 ARGUMENTS: EXPRESSIONS

Arguments for the opcodes and pseudo-ops are usually called expressions because they resemble mathematical expressions, such as 5+4*3. The parts of an expression are called operands (5, 4, and 3 in the mathematical expression) and operators (the + and * are examples). Expressions may contain one operand or more than one. One operand expressions are probably the form most commonly used as arguments. If the expression contains more than one operand, the operands are related to each other by an operator. For example:

```
5+4   6-3   7*2   8/7   9>8
```

and so on. In MACRO-80, operands are numeric values represented by numbers, characters, symbols, or 8080 opcodes. Operators may be arithmetic or logical.

You are probably familiar with the various forms of expressions that can be used as arguments, but you may want to review the details given below for characteristics unique to MACRO-80.

The following sections define the forms of operands and operators MACRO-80 supports.

3.4.1 Operands

Operands may be numbers, characters, symbols, or 8080 opcodes.

Numbers

The default base for numbers is decimal. The base may be changed by the .RADIX pseudo-op. Any base from 2 (binary) to 16 (hexadecimal) may be selected. When the radix is greater than 10, A-F are used for the digits following 9. If the first digit of a number is not numeric, the number must be preceded by a zero.

A number is always evaluated in the current radix unless one of the following special notations is used:

nnnnB	Binary
nnnnD	Decimal
nnnnO	Octal
nnnnH	Hexadecimal
X'nnnn'	Hexadecimal

Numbers are 16-bit unsigned binary quantities. Overflow of a number beyond two bytes (16 bits -- that is, 65535 decimal) is ignored, and the result is the low order 16 bits.

ASCII Strings

A string is composed of zero or more characters delimited by quotation marks. Either single (') or double (") quotation marks may be used as string delimiters. When a quoted string is entered as an argument, the values of the characters are stored in memory one after the other. For example:

```
DB      "ABC"
```

stores the ASCII value of A at the first address, B at the second address, and C at the third.

The delimiter quotes may be used as characters if they appear twice for every character occurrence desired. For example, the statement

```
"I am ""great"" today"
```

stores the string

```
I am "great" today
```

If no characters are placed between the quotation marks, the string is evaluated as a null string.

Character Constants

Like strings, character constants are composed of zero, one, or two ASCII characters, delimited by quotation marks. Either single or double quotation marks may be used as delimiters. The delimiter quotes may be used as characters if they appear twice for every character occurrence desired.

The differences are:

1. A character constant is only zero, one, or two characters.

2. Quoted characters are a character constant only if the expression has more than one operand. If the characters are entered as the only operand, they are evaluated and stored as a string. For example:

   'A'+1 is a character constant, but

   'A' is a string.

3. The value of a character constant is calculated, and the result is stored with the low-byte in the first address and the high-byte in the second. For example:

3. The value of a character constant is calculated, and the result is stored with the low-byte in the first address and the high-byte in the second. For example:

```
DW    'AB'+0
```

evaluates to 4142H and stores 42 in the first address and 41 in the second.

A character constant comprised of one character has as its value the ASCII value of that character. That is, the high order byte of the value is zero, and the low order byte is the ASCII value of the character. For example, the value of the constant 'A' is 41H.

A character constant comprised of two characters has as its value the ASCII value of the first character in the high order byte and the ASCII value of the second character in the low order byte. For example, the value of the character constant 'AB'+0 is 41H*256+42H+0.

The ASCII decimal and hexadecimal values for characters are listed in Appendix C.

Symbols in Expressions

A symbol may be used as an operand in an expression. The symbol is evaluated, and the value is substituted for the symbol. The Operation is performed using the symbol's value.

The benefit of using symbols as operands is that the programmer need not remember the exact value each time it is needed; rather, the symbol name can be used. The name is usually easier to remember, especially if the symbol name is made mnemonic. The use of symbols as operands becomes more attractive, of course, as the number of symbols in a program increases.

Rules Governing the Use of EXTERNALS in expressions:

1. EXTERNAL symbols may be used in expressions with the following operators only:

```
+    -    *    /    MOD    HIGH    LOW
```

2. If an EXTERNAL symbol is used in an expression, the result of the expression is always external.

MODE Rules affecting SYMBOLS in expressions:

1. In any operation, except AND, OR, or XOR, the operands may be any mode.

2. For AND, OR, XOR, SHL, and SHR, both operands must be absolute and internal.

3. When an expression contains an Absolute operand and an operand in another mode, the result of the expression will be in the other (not Absolute) mode.

4. When subtracting two operands in different modes, the result will be in Absolute mode. Otherwise, the result will be in the mode of the operands.

5. When adding a data relative symbol and a code relative symbol, the result will be unknown, and MACRO-80 passes the expression to LINK-80 as an unknown, which LINK-80 resolves.

Current Program Counter Symbol

One additional symbol for the Argument field only must be noted: the current program counter symbol. The current program counter is the address of the next instruction to be assembled. The current program counter is often a convenient reference point for calculating new addresses. Instead of remembering or calculating the current program address, the programmer uses a symbol that tells the assembler to use the value of the current program address.

The current program counter symbol is $.

8080 Opcodes as Operands

8080 opcodes are valid one-byte operands in 8080 mode only. During assembly, the opcode is evaluated to its hexadecimal value.

To use 8080 opcodes as operands, first set the .8080 pseudo-op. See the Language Set Selection Pseudo-ops section of Chapter 4 for a description of how to use the .8080 pseudo-op.

Only the first byte is a valid operand. Use parentheses to direct the assembler to generate one byte for opcodes that normally generate more than one. For example:

```
MVI     A,(JMP)
ADI     (CPI)
MVI     B,(RNZ)
CPI     (INX H)
ACI     (LXI B)
MVI     C,MOV A,B
```

The assembler returns an error if more than one byte is included in the operand (inside the parentheses) -- such as (CPI 5), (LXI B,LABEL1), or (JMP LABEL2).

Opcodes that generate one byte normally may be used as operands without being enclosed in parentheses.

3.4.2 Operators

MACRO-80 allows both arithmetic and logical operators. Operators which return true or false conditions return true if the result is any non-zero value and false if the result is zero.

The following arithmetic and logical operators are allowed in expressions.

Operator	Definition
NUL	Returns true if the argument (a parameter) is null. The remainder of the line after NUL is taken as the argument to NUL. The conditional IF NUL <argument> is false if the first character of the argument is anything other than a semicolon or carriage return. Note that IFB and IFNB perform the same functions but are simpler to use. (Refer to the Conditional Assembly Facility section in Chapter 4.)
TYPE	The TYPE operator returns a byte that describes two characteristics of its argument: 1) the mode, and 2) whether it is External or not. The argument to TYPE may be any expression (string, numeric, logical). If the expression is invalid, TYPE returns zero. The byte that is returned is configured as

follows:

The lower two bits are the mode. If the lower two bits are:

0 the mode is Absolute
1 the mode is Program Relative
2 the mode is Data Relative
3 the mode is Common Relative

The high bit (80H) is the External bit. If the high bit is on, the expression contains an External. If the high bit is off, the expression is local (not External).

The Defined bit is 20H. This bit is on if the expression is locally defined, and it is off if the expression is undefined or external. If neither bit is on, the expression is invalid.

TYPE is usually used inside macros, where an argument type may need to be tested to make a decision regarding program flow; for example, when conditional assembly is involved.

EXAMPLE:

```
FOO     MACRO    X
        LOCAL    Z
Z       SET TYPE X
IF      Z...
```

TYPE tests the mode and type of X. Depending on the evaluation of X, the block of code beginning with IF Z... may be assembled or omitted.

LOW — Isolates the low order 8 bits of an absolute 16-bit value.

HIGH — Isolates the high order 8 bits of an absolute 16-bit value.

* — Multiply

/ — Divide

MOD — Modulo. Divide the left operand by the right operand and return the value of the remainder (modulo).

SHR	Shift Right. SHR is followed by an integer which specifies the number of bit positions the value is to be right shifted.
SHL	Shift Left. SHL is followed by an integer which specifies the number of bit positions the value is to be left shifted.
- (Unary Minus)	Indicates that following value is negative, as in a negative integer.
+	Add
-	Subtract the right operand from the left operand.
EQ	Equal. Returns true if the operands equal each other.
NE	Not Equal. Returns true if the operands are not equal to each other.
LT	Less Than. Returns true if the left operand is less than the right operand.
LE	Less than or Equal. Returns true if the left operand is less than or equal to the right operand.
GT	Greater Than. Returns true if the left operand is greater than the right operand.
GE	Greater than or Equal. Returns true if the left operand is greater than or equal to the right operand.
NOT	Logical NOT. Returns true if left operand is true and right is false or if right is true and left is false. Returns false if both are true or both are false.
AND	Logical AND. Returns true if both operators are true. Returns false if either operator is false or if both are false. Both operands must be absolute values.
OR	Logical OR. Returns true if either operator is true or if both are true. Returns false if both operators are false. Both operands must be absolute values.

XOR — Exclusive OR. Returns true if either operator is true and the other is false. Returns false if both operators are true or if both operators are false. Both operands must be absolute values.

The order of precedence for the operators is:

NUL, TYPE

LOW, HIGH

*, /, MOD, SHR, SHL

Unary Minus

+, -

EQ, NE, LT, LE, GT, GE

NOT

AND

OR, XOR

Subexpressions involving operators of higher precedence than an expression are computed first. The order of precedence may be altered by using parentheses around portions of an expression you wish to give higher precedence.

All operators except +, -, *, and / must be separated from their operands by at least one space.

The byte isolation operators (HIGH and LOW) isolate the high- or low-order 8 bits of a 16-bit value.

Contents

CHAPTER 4

ASSEMBLER FEATURES

The MACRO-80 macro assembler features three general facilities: single-function pseudo-ops, a macro facility, and a conditional assembly facility.

4.1 SINGLE-FUNCTION PSEUDO-OPS

Single-function pseudo-ops involve only their own statement line and direct the assembler to perform only one function. (Macros and conditionals involve more than one line of code, so they may be thought of as block pseudo-ops.)

The Single-Function Pseudo-ops are divided into five types: Instruction Set Selection, Data Definition and Symbol Definition, PC Mode, File Related, and Listing Control.

INSTRUCTION SET SELECTION

The default instruction set mode is 8080. If the correct instruction set selection pseudo-op is not given, the assembler will return fatal errors for opcodes that are not valid for the current instruction set selection mode. That is, .Z80 assembles Z80 opcodes only; .8080 assembles 8080 opcodes only. Therefore, if you have written any assembly language programs for Z80, you need to insert the .Z80 instruction set pseudo-op at the beginning of the program file.

Note that all the pseudo-ops listed in this chapter will assemble in both instruction set modes.

.Z80

.Z80 takes no arguments. .Z80 directs MACRO-80 to assemble Z80 opcodes.

.8080

.8080 takes no arguments. .8080 directs MACRO-80 to assemble 8080 opcodes. (default)

All opcodes entered following an Instruction Set Selection pseudo-op will be assembled as that type of code until a different Instruction Set Selection pseudo-op is encountered.

If you enter an opcode not belonging to the selected instruction set, MACRO-80 will return an Objectionable Syntax error (letter O).

DATA DEFINITION AND SYMBOL DEFINITION

All of the data definition and symbol definition pseudo-ops are supported in both instruction set modes. (The one notable exception is SET, which is illegal in .Z80 mode. For your information, The following notation has been placed before the pseudo-op syntax to indicate which microprocessor the pseudo-op is usually associated with:

* indicates a Z80 pseudo-op

No asterisk indicates an Intel 8080 pseudo-op

Define Byte

```
    DB <exp>[,<exp>...]
*   DEFB <exp>[,<exp>...]
    DB <string>[<string>...]
*   DEFM <string>[,<string>...]
```

The arguments to DB are either expressions or strings. The arguments to DEFB are expressions. The arguments to DEFM are strings. Strings must be enclosed in quotes, either single or double.

NOTE: DB is used throughout the following explanation to represent all the Define Byte pseudo-ops.

DB is used to store a value (string or numeric) in a memory location, beginning with the current location counter.

Expressions must evaluate to one byte. (If the high byte of the result is 0 or 255, no error is given; otherwise, an A error results.)

Strings of three or more characters may not be used in expressions (i.e., they must be immediately followed by a comma or the end of the line). The characters in a 8080 or Z80 string are stored in the order of appearance, each as a one-byte value with the high order bit set to zero.

EXAMPLE:

```
                        DB      'AB'
                        DB      'AB' AND 0FFH
                        DB      'ABC'
```

assembles as:

```
0000'  41 42            DB      'AB'
0002'  42               DB      'AB' AND 0FFH
0003'  41 42 43         DB      'ABC'
```

Define Character

DC <string>

DC stores the characters in <string> in successive memory locations beginning with the current location counter. As with DB, characters are stored in order of appearance, each as a one-byte value with the high order bit set to zero. However, DC stores the last character of the string with the high order bit set to one. An error will result if the argument to DC is a null string.

EXAMPLE:

```
                    FOO:        DC          "ABC"
```

assembles to:

```
0000'  41 42 C3     FOO:        DC          "ABC"
```

Define Space

```
    DS   <exp>[,<val>]
*   DEFS <exp>[,<val>]
```

The define space pseudo-ops reserve an area of memory. The value of <exp> gives the number of bytes to be reserved.

To initialize the reserved space, set <val> to the value desired. If <val> is nul (that is, omitted), the reserved space is left as is (uninitialized); the reserved block of memory is not automatically initialized to zeros. As an alternative to setting <val> to zero, when you want the define space block initialized to zeros, you may use the /M switch at assembly time. See the Switches section in Chapter 5, Running MACRO-80, for a description of the /M switch.

All names used in <exp> must be previously defined (i.e., all names known at that point on pass 1). Otherwise, a V error is generated during pass 1, and a U error may be generated during pass 2. If a U error is not generated during pass 2, a phase error will probably be generated because the define space pseudo-op generated no code on pass 1.

EXAMPLE:

```
    DS      100H
```

reserves 100H bytes of memory, uninitialized (whatever values were in those bytes before the program was loaded will still be there). Use the /M switch at assembly time to initialized the 100H bytes to zero, if you want. Or, use the following statement to initialize a reserved space to zero or any other value:

```
    DS      100H,2
```

reserves 100H bytes, each initialized to a value of 2.

```
0000  1234      FOO:    DW      1234H
```

Note: The bytes are shown on the listing in the order entered, not the order stored.

Equate

<name> EQU <exp>

EQU assigns the value of <exp> to <name>. The <name> may be a label, a symbol, or a variable, and may be used subsequently in expressions. <name> may not be followed by colon(s).

If <exp> is External, an error is generated. If <name> already has a value other than <exp>, an M error is generated.

If you will want to redefine <name> later in the program, use the SET or ASET pseudo-op to define <name> instead of EQU.

Contrast with SET.

EXAMPLE:

```
BUF   EQU   0F3H
```

External Symbol

```
    EXT <name>[,<name>...]
    EXTRN <name>[,<name>...]
*   EXTERNAL <name>[,<name>...]
    BYTE EXT <symbol>
    BYTE EXTRN <symbol>
    BYTE EXTERNAL <symbol>
```

The External symbol pseudo-ops declare that the name(s) in the list are External (i.e., defined in a different module). If any item in the list refers to a name that is defined in the current program, an M error results. A reference to a name where the name is followed immediately be two pound signs (e.g., NAME##) also declares the name as External.

Externals may evaluate to either one or two bytes. For all External symbol names, only the first 6 characters are passed to the linker. Additional characters are truncated internally.

EXAMPLE:

```
    EXTRN     ITRAN     ;tranf init rtn
```

MACRO-80 will generate no code for this statement when this module is assembled. When ITRAN is used as an argument to a CALL statement, the CALL ITRAN statement generates the code for CALL but a zero value (0000*) for ITRAN. At link time, LINK-80 will search all modules loaded for a PUBLIC ITRAN statement and use the definition of ITRAN found in that module to define ITRAN in the CALL ITRAN statement.

Public Symbol

```
ENTRY <name>[,<name>...]
GLOBAL <name>[,<name>...]
PUBLIC <name>[,<name>...]
```

The Public symbol pseudo-ops declare each name in the list as internal and therefore available for use by this program and other programs to be loaded concurrently and linked with LINK-80. All of the names in the list must be defined in the current program, or a U error results. An M error is generated if the name is an External name or common block name.

Only the first 6 characters of a Public symbol name are passed to the linker. Additional characters are truncated internally.

EXAMPLE:

```
            PUBLIC  ITRAN   ;tranf init rtn
              .
              .
              .
ITRAN:      LD      HL,PASSA  ;store addr of
                              ;reg pass area
```

MACRO-80 assembles the LD statement as usuall but generates no code for the PUBLIC ITRAN statement. When LINK-80 sees EXTRN ITRAN in another module, it knows to search until it sees this PUBLIC ITRAN statement. Then, LINK-80 links the value of ITRAN: LD HL,PASSA statement to the CALL ITRAN statement in the other module(s).

Set

```
    <name> SET <exp>      (Not in .Z80 mode)
*   <name> DEFL <exp>
    <name> ASET <exp>
```

The Set pseudo-ops assign the value of <exp> to <name>. The <name> may be a label, a symbol, or a variable, and may be used subsequently in expressions. <name> may not be followed by colon(s). If <exp> is External, an error is generated.

The SET pseudo-op may not be used in .Z80 mode because SET is a Z80 opcode. Both ASET and DEFL may be used in both instruction set modes.

Use one of the SET pseudo-ops instead of EQU to define and redefine <name>s you may want to redefine later. <name> may be redefined with any of the Set pseudo-ops, regardless of which pseudo-op was used to define <name> originally (the prohibition against SET in .Z80 mode still applies, however).

Contrast with EQU.

EXAMPLE:

```
    FOO ASET BAZ+1000H
```

Whenever FOO is used as an expression (operand), the ALDS assembler will evaluate BAZ+1000H and substitute the value for FOO. Later, if you want FOO to represent a different value, simply reenter the FOO ASET statement with a different expression.

```
    FOO ASET BAZ+1000H
          .
          .
          .
    FOO ASET 3000H
          .
          .
          .
    FOO DEFL 6CDEH
```

PC MODE

Many of the pseudo-ops operate on or from the current location counter, also known as the program counter or PC. The current PC is the address of the next byte to be generated.

In MACRO-80, the PC has a mode, which gives symbols and expressions their modes. (Refer again to the Overview in Chapter 1 and the Symbols section in Chapter 3, if necessary.) Each mode is given a segment of memory by LINK-80 for the instructions assembled to each mode.

The four modes are Absolute, Data Relative, Code Relative, and COMMON Relative.

If the PC mode is absolute, the PC is an absolute address. If the PC mode is relative, the PC is a relative address and may be considered an offset from the absolute address where the beginning of that relative segment will be loaded by LINK-80.

The PC mode pseudo-ops are used to specify in which PC mode a segment of a program will be assembled.

Absolute Segment

ASEG

ASEG never has operands. ASEG generates non-relocatable code.

ASEG sets the location counter to an absolute segment (actual address) of memory. The ASEG will default to 0, which could cause the module to write over part of the operating system. We recommend that each ASEG be followed with an ORG statement set at 103H or higher.

Code Segment

CSEG

CSEG never has an operand. Code assembled in Code Relative mode can be loaded into ROM/PROM.

CSEG resets the location counter to the code relative segment of memory. The location will be that of the last CSEG (default is 0), unless an ORG is done after the CSEG to change the location.

Note, however, that the ORG statement does not set a hard (absolute) address under CSEG mode. An ORG statement under CSEG causes the assembler to add the number of bytes specified by the <exp> argument in the ORG statement to the last CSEG address loaded. If, for example, ORG 50 is given, MACRO-80 will add 50 bytes to the current CSEG location then begin loading the CSEG. The clearing effect of the ORG statement following CSEG (and DSEG as well) can be used to give the module an offset. The rationale for not allowing ORG to set an absolute address for CSEG is to keep the CSEG relocatable.

To set an absolute address for the CSEG, use the /P switch in LINK-80.

CSEG is the default mode of the assembler. Assembly begins with a CSEG automatically executed, and the location counter in the Code Relative mode, pointing to location 0 in the Code Relative segment of memory. All subsequent instructions will be assembled into the Code Relative segment of memory until an ASEG, DSEG, or COMMON pseudo-op is executed. CSEG is then entered to return the assembler to Code Relative mode, at which point the location counter returns to the next free location in the Code Relative segment.

Data Segment

DSEG

The DSEG pseudo-op never has operands. DSEG specifies segments of assembled relocatable code that will later be loaded into RAM only.

DSEG sets the location counter to the Data Relative segment of memory. The location of the data relative counter will be that of the last DSEG (default is 0), unless an ORG is done after the DSEG to change the location.

Note, however, that the ORG statement does not set a hard (absolute) address under DSEG mode. An ORG statement under DSEG causes the assembler to add the number of bytes specified by the <exp> argument in the ORG statement to the last DSEG address loaded. If, for example, ORG 50 is given, MACRO-80 will add 50 bytes to the last DSEG address loaded then begin loading the DSEG. The clearing effect of the ORG statement following DSEG (and CSEG as well) can be used to give the module an offset. The rationale for not allowing ORG to set an absolute address for DSEG is to keep the DSEG relocatable.

To set an absolute address for the DSEG, use the /D switch in LINK-80.

Common Block

COMMON /<block name>/

The argument to COMMON is the common block name. COMMON creates a common data area for every COMMON block that is named in the program. If <block name> is omitted or consists of spaces, the block is considered to be blank common.

COMMON statements are non-executable, storage allocating statements. .COMMON assigns variables, arrays, and data to a storage area called COMMON storage. This allows various program modules to share the same storage area. Statements entered following the .COMMON statement are assembled to the COMMON area under the <block name>. The length of a COMMON area is the number of bytes required to contain the variables, arrays, and data declared in the COMMON block, which ends when another PC mode pseudo-op is encountered. COMMON blocks of the same name may be different lengths. If the lengths differ, then the program module with the longest COMMON block must be loaded first (that is, must be the first module name given in the LINK-80 command line; see Chapter 6 for the description of LINK-80).

COMMON sets the location counter to the selected common block in memory. The location is always the beginning of the area so that compatibility with the FORTRAN COMMON statement is maintained.

EXAMPLE:

```
          COMMON /DATABIN/
ANVIL     EQU    100H
          DB     0FFH
          DW     1234H
          DCI    'FORGE'
          CSEG
             .
             .
             .
```

Set Origin

ORG <exp>

At any time, the value of a location counter may be changed by use of ORG. Under the ASEG PC mode, the location counter is set to the value of <exp>, and the assembler assigns generated code starting with that value. Under the CSEG, DSEG, and COMMON PC modes, the location counter for the segment is incremented by the value of <exp>, and the assembler assigns generated code starting with the value of that last segment address loaded plus the value of <exp>. All names used in <exp> must be known on pass 1, and the value must either be Absolute or in the same area as the location counter.

EXAMPLE:

```
DSEG
ORG     50
```

sets the Data Relative location counter to 50, relative to the start of the Data Relative segment of memory. This means that the first 50H addresses will be filled with 0. This method provides relocatability. The ORG <exp> statement does not specify a fixed address in CSEG or DSEG mode; rather, LINK-80 loads the segment at a flexible address appropriate to the modules being loaded together.

On the other hand, a program that begins with the statements

```
ASEG
ORG     800H
```

and is assembled entirely in Absolute mode will always load beginning at 800H, unless the ORG statement is changed in the source file. That is, ORG <exp> following ASEG originates the segment at a fixed (i.e., absolute) address specified by <exp>. However, the same program, assembled in Code Relative mode with no ORG statement, may be loaded at any specified address by appending the /P:<address> switch to the LINK-80 command string. (For details, see Section 6.3, Switches.)

Relocate

```
.PHASE <exp>
   .
   .
   .
.DEPHASE
```

.PHASE allows code to be located in one area, but executed only at a different area with a start address specified by <exp>. The <exp> must be an absolute value. .DEPHASE is used to indicate the end of the relocated block of code.

The PC mode within a .PHASE block is absolute, the same as the mode of the <exp> in the .PHASE statement. The code, however, is loaded in the area in effect when the .PHASE statement is encountered. The code within the block is later moved to the address specified by <exp> for execution.

EXAMPLE:

```
                        .PHASE      100H
                FOO:    CALL        BAZ
                        JMP         ZOO
                BAZ:    RET
                        .DEPHASE
                ZOO:    JMP         5
```

assembles as:

```
                        .PHASE      100H
0100    CD 0106 FOO:    CALL        BAZ
0103    C3 0007'        JMP         ZOO
0106    C9      BAZ:    RET
                        .DEPHASE
0007'   C3 0005 ZOO:    JMP         5
                        END
```

.PHASE....DEPHASE blocks are a way to execute a block of code at a specific absolute address.

FILE RELATED

The file related pseudo-ops insert long comments in the program, give the module a name, end the module, or move other files into the current program.

Comment

.COMMENT <delim><text><delim>

The first non-blank character encountered after .COMMENT is taken as the delimiter. The <text> following the delimiter becomes a comment block which continues until the next occurrence of <delimiter>.

Use the .COMMENT pseudo-op to make long comments. It is not necessary to enter the semicolon to indicate a COMMENT. Indeed, the main reason for using .COMMENT is to override the need to begin each comment line with a semicolon. During assembly, .COMMENT blocks are ignored and not assembled.

EXAMPLE:

```
.COMMENT * any amount of text
entered here
    .
    .
    .* ;return to normal assembly
```

End of Program

END [<exp>]

The END statement specifies the end of the module. If the END statement is not included, a %No END statement warning error message results.

The <exp> may be a label, symbol, number, or any other legal argument that LINK-80 can load as the starting point into the first address to be loaded. If <exp> is present, LINK-80 will place an 8080 JMP instruction at 0100H to the address of <exp>. If <exp> is not present, then no start address is passed to LINK-80 for that program, and execution begins at the first module loaded. (Also, if <exp> is not specified, the LINK-80 /G switch will not work for the module.)

The <exp> tells LINK-80 that the program is a main program. Without <exp>, LINK-80 takes assembly language programs as subroutines. If you link only assembly language programs and none contains an END statement with <exp>, LINK-80 will ask for a main program. If you link two or more programs with END <exp> statements, LINK-80 cannot distinguish which should be the main program.

If you want to link two or more main programs, use the /G:Name or /E:Name switches in LINK-80 (see Section 6.2.2, Switches). The "Name" will be the <exp> of the END statement for the program you want to serve as the main program.

If any high-level language program is loaded with assembly language modules, LINK-80 takes the high-level language program as the main program automatically. Therefore, if you want an assembly language module executed before the high-level language program, use the /G:Name or /E:Name switch in LINK-80 to set the assembly language module as the beginning of the program.

As an alternative, we recommend that you place a CALL or INCLUDE statement at the beginning of the high-level language program, and call in the assembly language program for execution prior to execution of the high-level language program.

Include

```
INCLUDE <filename>
$INCLUDE <filename>
MACLIB <filename>
```

All three pseudo-ops are synonomous.

These Include pseudo-ops insert source code from an alternate assembly language source file into the current source file during assembly. Use of an Include pseudo-op eliminates the need to repeat an often-used sequence of statements in the current source file.

The <filename> is any valid file specification for the operating system. If the filename extension and/or device designation are other than the default, source filename specifications must include them. The default filename extension for source files is .MAC. The default device designation is the currently logged drive or device.

The included file is opened and assembled into the current source file immediately following the Include pseudo-op statement. When end-of-file is reached, assembly resumes with the next statement following Include pseudo-op.

Nested Includes are not allowed. If encountered, they will result in an objectionable syntax error, O.

The file specified in the operand field must exist. If the file is not found, the error V (value error) is returned, and the Include is ignored. The V error is also returned if the Include filename extension is not .MAC.

On a MACRO-80 listing, the letter C is printed between the assembled code and the source line on each line assembled from an included file. See the Listing Control Pseudo-op section below for a description of listing file formats.

Name Module

NAME ('modname')

Name defines a name for the module. The parentheses and quotation marks around modname are required. Only the first six characters are significant in a module name.

A module name may also be defined with the TITLE pseudo-op. In the absence of both the NAME and TITLE pseudo-ops, the module name is created from the source filename.

Radix

.RADIX <exp>

The <exp> in a .RADIX statement is always a decimal numeric constant, regardless of the current radix.

The default input radix (or base) for all constants is decimal. The .RADIX pseudo-op allows you to change the input radix to any base in the range 2 to 16.

.RADIX does not change the radix of the listing; rather, it allows you to input numeric values in the radix you choose without special notation. (Values in other radixes still require the special notations described in Section 3.4.1.) Values in the generated code remain in hexadecimal radix.

EXAMPLE:

```
                    DEC:    DB      20
                            .RADIX  2
                    BIN:    DB      00011110
                            .RADIX  16
                    HEX:    DB      0CF
                            .RADIX  8
                    OCT:    DB      73
                            .RADIX  10
                    DECI:   DB      16
                    HEXA:   DB      OCH
```

assembles as:

```
0000'  14           DEC:    DB      20
0002                        .RADIX  2
0001'  1E           BIN:    DB      00011110
0010                        .RADIX  16
0002'  CF           HEX:    DB      0CF
0008                        .RADIX  8
0003'  3B           OCT:    DB      73
000A                        .RADIX  10
0004'  10           DECI:   DB      16
0005'  0C           HEXA:   DB      OCH
```

Request

.REQUEST <filename>[,<filename>...]

When you run LINK-80, .REQUEST sends a request to the LINK-80 linking loader to search the filenames in the list for undefined external symbols. If LINK-80 finds any undefined external symbols (external symbols for which a corresponding PUBLIC symbol is not currently loaded), you will know that you need to load one or more additional modules to complete linking.

The filenames in the list should be in the form of legal symbols. <filename> should not include a filename extension or device designation. LINK-80 assumes the default extension (.REL) and the currently logged disk drive.

EXAMPLE:

```
        .
        .
        .
    .REQUEST   SUBR1
        .
        .
        .
```

LINK-80 will search SUBR1 for external symbols which do not have corresonding PUBLIC symbol definitions declared among the currently loaded modules.

LISTING

Listing pseudo-ops perform two general functions: format control and listing control. Format control pseudo-ops allow the programmer to insert page breaks and direct page headings. Listing control pseudo-ops turn on and off the listing of all or part of the assembled file.

Format Control

These pseudo-ops allow you to direct page breaks, titles, and subtitles on your program listings.

Form Feed

```
*  *EJECT [<exp>]
   PAGE <exp>
   $EJECT
```

The form feed pseudo-ops cause the assembler to start a new output page. The assembler puts a form feed character in the listing file at the end of the page.

The value of <exp>, if included, becomes the new page size (measured in lines per page) and must be in the range 10 to 255. The default page size is 50 lines per page.

EXAMPLE:

```
        .
        .
        .
    *EJECT 58
        .
        .
        .
```

The assembler causes the printer to start a new page every time 58 lines of program have been printed.

Title

TITLE <text>

TITLE specifies a title to be listed on the first line of each page. If more than one TITLE is given, a Q error results. The first six characters of the title are used as the module name, unless a NAME pseudo-op is used. (If neither a TITLE nor a NAME pseudo-op is used, the module name is created from the source filename.)

EXAMPLE:

```
TITLE PROG1
   .
   .
   .
```

The module name is now PROG1. The module may be called by this name, which will be printed at the top of every listing page.

Subtitle

```
SUBTTL <text>
$TITLE ('<text>')
```

SUBTTL specifies a subtitle to be listed in each page heading on the line after the title. The <text> is truncated after 60 characters.

Any number of SUBTTLs may be given in a program. Each time the assembler encounters SUBTTL, it replaces the <text> from the previous SUBTTL with the <text> from the most recently encountered SUBTTL. To turn off SUBTTL for part of the output, enter a SUBTTL with a null string for <text>.

EXAMPLE:

```
    SUBTTL SPECIAL I/O ROUTINE
        .
        .
        .
    SUBTTL
        .
        .
        .
```

The first SUBTTL causes the subtitle SPECIAL I/O ROUTINE to be printed at the top of every page. The second SUBTTL turns off subtitle (the subtitle line on the listing is left blank).

General Listing Control

.LIST - List all lines with their code
.XLIST - Suppress all listing

.LIST is the default condition. If you specify a listing file in the command line, the file will be listed.

When .XLIST is encountered in the source file, source and object code will not be listed. .XLIST remains in effect until a .LIST is encountered.

.XLIST overrides all other listing control pseudo-ops. So, nothing will be listed, even if another listing pseudo-op (other than .LIST) is encountered.

EXAMPLE:

```
      .
      .
      .
    .XLIST    ;listing suspended here
      .
      .
      .
    .LIST     ;listing resumes here
```

Print At Terminal

.PRINTX <delim><text><delim>

The first non-blank character encountered after .PRINTX is the delimiter. The following text is listed on the terminal during assembly until another occurrence of the delimiter is encountered. .PRINTX is useful for displaying progress through a long assembly or for displaying the value of conditional assembly switches.

.PRINTX will output on both passes. If only one printout is desired, use the IF1 or IF2 pseudo-op, depending on which pass you want displayed. See the Conditional pseudo-ops for IF1 and IF2.

EXAMPLE:

```
    .PRINTX *Assembly half done*
```

The assembler will send this message to the terminal screen when encountered.

```
IF1
.PRINTX *Pass 1 done*    ;pass 1 message only
ENDIF

IF2
.PRINTX *Pass 2 done*    ;pass 2 message only
ENDIF
```

Conditional Listing Control

The three conditional listing control pseudo-ops are used to specify whether or not you wish statements contained within a false conditional block to appear on the listing. See also the description of the /X switch in Chapter 5.

Suppress False Conditionals

.SFCOND

.SFCOND suppresses the portion of the listing that contains conditional expressions that evaluate as false.

List False Conditionals

.LFCOND

.LFCOND assures the listing of conditional expressions that evaluate false.

Toggle False Listing Conditional

.TFCOND

.TFCOND toggles the current setting. .TFCOND operates independently from .LFCOND and .SFCOND. .TFCOND toggles the default setting, which is set by the presence or absence of the /X switch in the assembler command line. When /X is present, .TFCOND will cause false conditionals to list. When /X is not given, .TFCOND will suppress false conditionals.

Macro Expansion Listing Control

Expansion listing pseudo-ops control the listing of lines inside macro and repeat pseudo-op (REPT, IRP, IRPC) blocks, and may be used only inside a macro or repeat block.

Exclude Non-code Macro Lines

.XALL

.XALL is the default.

.XALL lists source code and object code produced by a macro, but source lines which do not generate code are not listed.

List Macro Text

.LALL

.LALL lists the complete macro text for all expansions, including lines that do not generate code.

Suppress Macro Listing

.SALL

.SALL suppresses listing of all text and object code produced by macros.

CREF Listing Control Pseudo-ops

You may want the option of generating a cross reference listing for part of a program but not all of it. To control the listing or suppressing of cross references, use the cross reference listing control pseudo-ops, .CREF and .XCREF, in the source file for MACRO-80. These two pseudo-ops may be entered at any point in the program in the OPERATOR field. Like the other listing control pseudo-ops, .CREF and .XCREF support no ARGUMENTs.

Suppress Cross References

.XCREF

.XCREF turns of the .CREF (default) pseudo-op. .XCREF remains in effect until MACRO-80 encounters .CREF. Use .XCREF to suppress the creation of cross references in selected portions of the file. Because neither .CREF nor .XCREF takes effect until the /C switch is set in the MACRO-80 command line, there is no need to use .XCREF if you want the usual List file (one without cross references); simply omit /C from the ALDS assembler command line.

List Cross References

.CREF

.CREF is the default condition. Use .CREF to restart the creation of a cross reference file after using the .XCREF pseudo-op. .CREF remains in effect until MACRO-80 encounters .XCREF. Note, however, that .CREF has no effect until the /C switch is set in the MACRO-80 command line.

4.2 MACRO FACILITY

The macro facility allows you to write blocks of code which can be repeated without recoding. The blocks of code begin with either the macro definition pseudo-op or one of the repetition pseudo-ops and end with the ENDM pseudo-op. All of the macro pseudo-ops may be used inside a macro block. In fact, nesting of macros is limited only by memory.

The macro facility of the MACRO-80 macro assembler includes pseudo-ops for:

macro definition:
- MACRO

repetitions:
- REPT (repeat)
- IRP (indefinite repeat)
- IRPC (indefinite repeat character)

termination:
- ENDM
- EXITM

unique symbols within macro blocks:
- LOCAL

The macro facility also supports some special macro operators:

- &
- ;;
- !
- %

Macro Definition

```
<name> MACRO <dummy>[,<dummy>...]
         .
         .
         .
       ENDM
```

The block of statements from the MACRO statement line to the ENDM statement line comprises the body of the macro, or the macro's definition.

<name> is like a LABEL and conforms to the rules for forming symbols. Note that <name> may be any length, but only the first 16 characters are passed to the linker. After the macro has been defined, <name> is used to invoke the macro.

A <dummy> is a place holder that is replaced by a parameter in a one-for-one text substitution when the MACRO block is used. Each <dummy> may be up to 32 characters long. The number of dummys is limited only by the length of a line. If you specify more than one dummy, they must be separated by commas. MACRO-80 interprets all characters between commas as a single dummy.

NOTE

A dummy is always recognized exclusively as a dummy. Even if a register name (such as A or B) is used as a dummy, it will be replaced by a parameter during expansion.

A macro block is not assembled when it is encountered. Rather, when you call a macro, the assembler "expands" the macro call statement by bringing in and assembling the appropriate macro block.

If you want to use the TITLE, SUBTTL, or NAME pseudo-ops for the portion of your program where a macro block appears, you should be careful about the form of the statement. If, for example, you enter SUBTTL MACRO DEFINITIONS, MACRO-80 will assemble the statement as a macro definition with SUBTTL as the macro name and DEFINITIONS as the dummy. To avoid this problem, alter the word MACRO in some way; e.g., - MACRO, MACROS, and so on.

Calling a Macro

To use a macro, enter a macro call statement:

<name> <parameter>[,<parameter>...]

<name> is the <name> of the MACRO block. A <parameter> replaces a <dummy> on a one-for-one basis. The number of parameters is limited only by the length of a line. If you enter more than one parameter, they must be separated by commas. If you place angle brackets around parameters separated by commas, the assembler will pass all the items inside the angle brackets as a single parameter. For example:

FOO 1,2,3,4,5

passes five parameters to the macro, but:

FOO <1,2,3,4,5>

passes only one.

The number of parameters in the macro call statement need not be the same as the number of dummys in the MACRO definition. If there are more parameters than dummys, the extras are ignored. If there are fewer, the extra dummys will be made null. The assembled code will include the macro block after each macro call statement.

EXAMPLE:

```
EXCHNG    MACRO    X,Y
          PUSH     X
          PUSH     Y
          POP      X
          POP      Y
          ENDM
```

If you then enter as part of a program some code and a macro call statement:

```
          LDA      2FH
          MOV      HL,A
          LDA      3FH
          MOV      DE,A
          EXCHNG   HL,DE
```

assembly generates the code:

```
0000'  3A 002F             LDA     2FH
0003'  67                  MOV     HL,A
0004'  3A 003F             LDA     3FH
0007'  57                  MOV     DE,A
                           EXCHNG  HL,DE
0008'  E5          +       PUSH    HL
0009'  D5          +       PUSH    DE
000A'  El          +       POP     HL
000B'  Dl          +       POP     DE
```

Repeat Pseudo-ops

The pseudo-ops in this group allow the operations in a block of code to be repeated for the number of times you specify. The major differences between the Repeat pseudo-ops and MACRO pseudo-op are:

1. MACRO gives the block a name by which to call in the code wherever and whenever needed; the macro block can be used in many different programs by simply entering a macro call statement.

2. MACRO allows parameters to be passed to the MACRO block when a MACRO is called; hence, parameters can be changed.

Repeat pseudo-op parameters must be assigned as a part of the code block. If the parameters are known in advance and will not change, and if the repetition is to be performed for every program execution, then Repeat pseudo-ops are convenient. With the MACRO pseudo-op, you must call in the MACRO each time it is needed.

Note that each Repeat pseudo-op must be matched with the ENDM pseudo-op to terminate the repeat block.

Repeat

```
REPT <exp>
  .
  .
  .
ENDM
```

Repeat block of statements between REPT and ENDM <exp> times. <exp> is evaluated as a 16-bit unsigned number. If <exp> contains an External symbol or undefined operands, an error is generated.

EXAMPLE:

```
                X    SET   0
                     REPT  10     ;generates DB 1 - DB 10
                X    SET   X+1
                     DB    X
                     ENDM
```

assembles as:

```
0000            X    SET   0
                     REPT  10     ;generates DB 1 - DB 10
                X    SET   X+1
                     DB    X
                     ENDM
0000'  01       +    DB    X
0001'  02       +    DB    X
0002'  03       +    DB    X
0003'  04       +    DB    X
0004'  05       +    DB    X
0005'  06       +    DB    X
0006'  07       +    DB    X
0007'  08       +    DB    X
0008'  09       +    DB    X
0009'  0A       +    DB    X
                     END
```

Indefinite Repeat

```
IRP <dummy>,<parameters inside angle brackets>
 .
 .
 .
ENDM
```

Parameters must be enclosed in angle brackets. Parameters may be any legal symbol, string, numeric, or character constant. The block of statements is repeated for each parameter. Each repetition substitutes the next parameter for every occurrence of <dummy> in the block. If a parameter is null (i.e., <>), the block is processed once with a null parameter.

EXAMPLE:

```
    IRP     X,<1,2,3,4,5,6,7,8,9,10>
    DB      X
    ENDM
```

This example generates the same bytes (DB 1 - DB 10) as the REPT example.

When IRP is used inside a MACRO definition block, angle brackets around parameters in the macro call statement are removed before the parameters are passed to the macro block. An example, which generates the same code as above, illustrates the removal of one level of brackets from the parameters:

```
    FOO     MACRO     X
            IRP       Y,<X>
            DB        Y
            ENDM
            ENDM
```

When the macro call statement

```
    FOO <1,2,3,4,5,6,7,8,9,10>
```

is assembled, the macro expansion becomes:

```
    IRP       Y,<1,2,3,4,5,6,7,8,9,10>
    DB        Y
    ENDM
```

The angle brackets around the parameters are removed, and all items are passed as a single parameter.

Indefinite Repeat Character

```
IRPC <dummy>,<string>
  .
  .
  .
ENDM
```

The statements in the block are repeated once for each character in the string. Each repetition substitutes the next character in the string for every occurrence of <dummy> in the block.

EXAMPLE:

```
    IRPC    X,0123456789
    DB      X+1
    ENDM
```

This example generates the same code (DB 1 - DB 10) as the two previous examples.

Termination

End Macro

ENDM

ENDM tells the assembler that the MACRO or Repeat block is ended.

Every MACRO, REPT, IRP, and IRPC must be terminated with the ENDM pseudo-op. Otherwise, the 'Unterminated REPT/IRP/IRPC/MACRO' message is generated at the end of each pass. An unmatched ENDM causes an O error.

If you wish to be able to exit from a MACRO or repeat block before expansion is completed, use EXITM.

Exit Macro

EXITM

The EXITM pseudo-op is used inside a MACRO or Repeat block to terminate an expansion when some condition makes the remaining expansion unnecessary or undesirable. Usually EXITM is used in conjunction with a conditional pseudo-op.

When an EXITM is assembled, the expansion is exited immediately. Any remaining expansion or repetition is not generated. If the block containing the EXITM is nested within another block, the outer level continues to be expanded.

EXAMPLE:

```
FOO     MACRO     X
Y       SET       0
        REPT      X
Y       SET       Y+1
        IFE       Y-0FFH  ;test Y
        EXITM     ;if true, exit REPT
        ENDIF
        DB        Y
        ENDM
        ENDM
```

Macro Symbol

LOCAL <dummy>[,<dummy>...]

The LOCAL pseudo-op is allowed only inside a MACRO definition block. When LOCAL is executed, the assembler creates a unique symbol for each <dummy> and substitutes that symbol for each occurrence of the <dummy> in the expansion. These unique symbols are usually used to define a label within a macro, thus eliminating multiple-defined labels on successive expansions of the macro. The symbols created by the assembler range from ..0001 to ..FFFF. Users should avoid the form ..nnnn for their own symbols. A LOCAL statement must precede all other types of statements in the macro definition.

EXAMPLE:

```
                    FOO         MACRO   NUM,Y
                                LOCAL   A,B,C,D,E
                    A:          DB      7
                    B:          DB      8
                    C:          DB      Y
                    D:          DB      Y+1
                    E:          DW      NUM+1
                                JMP     A
                                ENDM
                                FOO     0C00H,0BEH
                                END
```

generates the following code (notice that MACRO-80 has substituted LABEL names in the form ..nnnn for the instances of the dummy symbols):

```
                    FOO         MACRO   NUM,Y
                                LOCAL   A,B,C,D,E
                    A:          DB      7
                    B:          DB      8
                    C:          DB      Y
                    D:          DB      Y+1
                    E:          DW      NUM+1
                                JMP     A
                                ENDM
                                FOO     0C00H,0BEH
0000'  07           +..0000:    DB      7
0001'  08           +..0001:    DB      8
0002'  BE           +..0002:    DB      0BEH
0003'  BF           +..0003:    DB      0BEH+1
0004'  0C01         +..0004:    DW      0C00H+1
0006'  C3 0000'     +           JMP     ..0000
                                END
```

Special Macro Operators

Several special operators can be used in a macro block to select additional assembly functions.

& Ampersand concatenates text or symbols. (The & may not be used in a macro call statement.) A dummy parameter in a quoted string will not be substituted in expansion unless preceded immediately by &. To form a symbol from text and a dummy, put & between them.

For example:

```
ERRGEN      MACRO    X
ERROR&X:    PUSH     B
            MVI      B,'&X'
            JMP      ERROR
            ENDM
```

The call ERRGEN A will then generate:

```
ERRORA:     PUSH     B
            MVI      B,'A'
            JMP      ERROR
```

;; In a block operation, a comment preceded by two semicolons is not saved as a part of the expansion (i.e., it will not appear on the listing even under .LALL). A comment preceded by only one semicolon, however, will be preserved and appear in the expansion.

! An exclamation point may be entered in an argument to indicate that the next character is to be taken literally. Therefore, !; is equivalent to <;>.

% The percent sign is used only in a macro argument to convert the expression that follows it (usually a symbol) to a number in the current radix (set by the .RADIX pseudo-op). During macro expansion, the number derived from converting the expression is substituted for the dummy. Using the % special operator allows a macro call by value. (Usually, a macro call is a call by reference with the text of the macro argument substituting exactly for the dummy.)

The expression following the % must conform to the same rules as expressions for the DS (Define Space) pseudo-op. That is, a valid expression that evaluates to an absolute (non-relocatable) constant is required.

EXAMPLE:

```
PRINTE    MACRO     MSG,N
           .PRINTX  * MSG,N *
          ENDM
SYM1      EQU       100
SYM2      EQU       200
          PRINTE <SYM1 + SYM2 = >,%(SYM1 + SYM2)
```

Normally, the macro call statement would cause the string (SYM1 + SYM2) to be substituted for the dummy N. The result would be:

```
    .PRINTX   * SYM1 + SYM2 = (SYM1 + SYM2)
```

When the % is placed in front of the parameter, the assembler generates:

```
    .PRINTX   * SYM1 + SYM2 = 300 *
```

Each IFxxxx must have a matching ENDIF to terminate the conditional. Each COND must have a matching ENDC to terminate the conditional. Otherwise, an 'Unterminated conditional' message is generated at the end of each pass. An ENDIF without a matching IF or an ENDC without a matching COND causes a C error.

The assembler evaluates the conditional statement to TRUE (which equals FFFFH, or -1, or any non-zero value), or to FALSE (which equals 0000H). The code in the conditional block is assembled if the evaluation matches the condition defined in the conditional statement. If the evaluation does not match, the assembler either ignores the conditional block completely or, if the conditional block contains the optional ELSE statement, assembles only the ELSE portion.

Conditionals may be nested up to 255 levels. Any argument to a conditional must be known on pass 1 to avoid V errors and incorrect evaluation. For IF/IFT/COND and IFF/IFE the expression must involve values which were previously defined, and the expression must be Absolute. If the name is defined after an IFDEF or IFNDEF, pass 1 considers the name to be undefined, but it will be defined on pass 2.

Each conditional block may include the optional ELSE pseudo-op, which allows alternate code to be generated when the opposite condition exists. Only one ELSE is permitted for a given IFxxxx/COND. An ELSE is always bound to the most recent, open IF. A conditional with more than one ELSE or an ELSE without a conditional will cause a C error.

Conditional Pseudo-ops

```
    IF <exp>
    IFT <exp>
*   COND <exp>
```

If <exp> evaluates to not-0, the statements within the conditional block are assembled.

```
IFE <exp>
IFF <exp>
```

If <exp> evaluates to 0, the statements in the conditional block are assembled.

IF1 Pass 1 Conditional

If the assembler is in pass 1, the statements in the conditional block are assembled.

IF2 Pass 2 Conditional

If the assembler is in pass 2, the statements in the conditional block are assembled.

IFDEF <symbol>

If the <symbol> is defined or has been declared External, the statements in the conditional block are assembled.

IFNDEF <symbol>

If the <symbol> is not defined or not declared External, the statements in the conditional block are assembled.

IFB <arg>

The angle brackets around <arg> are required.

If the <arg> is blank (none given) or null (two angle brackets with nothing in between, <>), the statements in the conditional block are assembled.

IFNB <arg>

The angle brackets around <arg> are required.

If <arg> is not blank, the statements in the conditional block are assembled. Used for testing for dummy parameters.

IFIDN <arg1>,<arg2>

The angle brackets around <arg1> and <arg2> are required.

If the string <arg1> is identical to the string <arg2>, the statements in the conditional block are assembled.

IFDIF <arg1>,<arg2>

The angle brackets around <arg1> and <arg2> are required.

If the string <arg1> is different from the string <arg2>, the statements in the conditional block are assembled.

ELSE

The ELSE pseudo-op allows you to generate alternate code when the opposite condition exists. May be used with any of the conditional pseudo-ops.

ENDIF
* ENDC

These pseudo-ops terminate conditional blocks. A terminate pseudo-op must be given for every conditional pseudo-op used. ENDIF must be matched with an IFxxxx pseudo-op. ENDC must be matched with the COND pseudo-op.

Contents

CHAPTER 5

RUNNING MACRO-80

When you have completed creating the assembly language source file, you are ready to assemble it. MACRO-80 assembles the source file statements, including expanding macros and repeat pseudo-ops. The result of assembly will be relocatable object code which is ready to link and load with LINK-80. The relocatable object code can be saved in a disk file, which the assembler gives the filename extension .REL. The assembled (REL) file is not an executable file. The file will be executable only after it is processed through LINK-80.

MACRO-80 resides in approximately 19K of memory and has an assembly rate of over 1000 lines per minute. MACRO-80 runs under the CP/M operating system.

MACRO-80 assembles your source file in two passes. During pass 1, MACRO-80 evaluates the program statements, calculates how much code it will generate, builds a symbol table where all symbols are assigned values, and expands macro call statements. During pass 2, MACRO-80 fills in the symbol and expression values from the symbol table, again expands macro call statements, and emits the relocatable code. MACRO-80 checks the values of symbols, expressions, and macros during both passes. If a value during pass 2 is different from the value during pass 1, MACRO-80 returns a phase error code.

Before MACRO-80 can be run, the diskette which contains MACRO-80 must be inserted in the appropriate disk drive. The diskette on which you created the source file must also be in a disk drive.

5.1 INVOKING MACRO-80

To invoke MACRO-80, enter:

```
M80
```

The program file M80.COM will be loaded. MACRO-80 will display an asterisk (*) to indicate that the assembler is ready to accept a command line.

5.2 MACRO-80 COMMAND LINE

The command line for MACRO-80 consists of four fields, labeled:

```
Object,List=Source/Switch
```

The command line may be entered on its own line, or it may be entered at the same time as the M80 command. (If M80 and the command line are entered on one line, MACRO-80 will not return the asterisk prompt.) Entering the command line on its own line allows single drive configurations to use MACRO-80. In addition, by entering M80 and the command line separately, you are able to perform another assembly without reinvoking MACRO-80. When assembly is finished, MACRO-80 will return the asterisk (*) prompt and wait for another command line. To exit MACRO-80 when you have entered M80 and the command line separately, type <CTRL-C>.

If you are performing only one assembly, entering the command line on the same line as M80 is convenient; it requires less typing and allows the assembly operation to be part of a SUBMIT command. When you enter M80 and the command line together, MACRO-80 exits automatically to the operating system.

NOTE

If you enter M80 and the command line separately, you must enter the command line in upper case only. If you do not, MACRO-80 will return a ?Command Error message. If you enter M80 and the command line on one line, the entries may be in either upper or lower case (or mixed) because CP/M converts all entries to upper case before passing the entries.

Source (=filename)

To assemble your source program, you must enter at least an equal sign (=) and the source filename.

The =filename indicates which source file you want to assemble. If the source file disk is not in the currently logged drive, you must include the drive designation as part of the filename. If the source filename is entered without an extension, MACRO-80 assumes that the extension is .MAC. If the extension is not .MAC, you must include the extension as part of the filename. For other possibilities for drive/device designations and filename extensions, see the Additional Command Line Entries section, below.)

The Source entry is the only entry required besides M80.

The simplest command is:

```
M80 =Source
```

This command directs MACRO-80 to assemble the source file and save the result in a relocatable object file (called a REL file) with the same name as the source file. If the source file is NEIL.MAC, the command line:

```
M80 =NEIL
```

generates an assembled file named NEIL.REL.

An additional option is to enter only a comma (,) to the left of the equal sign. When MACRO-80 sees a comma as the first entry after the M80 entry, it suppresses all output files (Object and List). The command line

```
M80 ,=NEIL
```

causes MACRO-80 to assemble the file NEIL.MAC, but no output files are created. Programmers use this command line to check syntax in the source program before saving the assembled program. Because no files are generated, the assembly is completed faster and errors are known sooner.

Object (filename)

An Object entry is always optional. However, certain circumstances will compel you to make some entry for the Object.

The Object file saves the assembled program in a disk file. LINK-80 uses the Object file to create an executable program. If both Object and List entries are omitted from a command line (as in =Source), MACRO-80 will generate an Object file with the same filename as the Source, but with the default extension .REL.

If you want your Object file to have a name different from the source file, you must enter a filename in the Object field. MACRO-80 will still append the filename extension .REL, unless you also enter an extension.

Also, if you want both a List file and a REL file generated, you must enter a filename for the Object, even if you want the REL file named after the source file. If you enter a filename for the List but omit the Object, no REL file will be generated. Programmers do use this feature for checking the program for errors before final assembly. The program listing aids debugging.

The name for the Object file may be the same as the source filename or any other legal filename you choose. Since it is practical to have all files which relate to a program carry some mutual indication of their relationship, most often you will want to give your object file the same name as your source file.

List (,filename)

A List entry is always optional. The comma is required in front of all List entries. If you want a List file, enter a ,filename for the List. (There is an alternative to this rule. See the Switches section below for discussion of the /L switch.)

MACRO-80 appends the default extension .PRN to the List file unless you specify a different extension in the List entry.

The command line:

```
M80 ,NEIL=NEIL
```

assembles the file NEIL.MAC (source file) and creates the List file NEIL.PRN. An Object (REL) file is not created.

The name may be the same as the source filename or any other legal filename you choose. Since it is practical to have all files which relate to a program carry some mutual indication of their relationship, most often you will want to give your listing file the same name as your source file.

Avoid entering only a comma for the List after entering a filename for the Object. For example:

```
M80 NEIL,=NEIL
```

MACRO-80 will probably ignore the comma and assemble the source file into a REL file. It is possible that MACRO-80 might return a COMMAND ERROR message.

If you enter only a comma for the List and nothing for the Object, MACRO-80 will assemble the source file, but will generate no output files. This command

```
M80 ,=Source
```

allows you to check the source program for syntax errors before saving the assembled program in a disk file. While MACRO-80 always checks for errors, this command form provides much faster assembly because the output files do not have to be created.

At the end of assembly, MACRO-80 will print the message:

```
[xx][No] Fatal errors [,xx warnings]
```

This message reports the number of fatal errors and warning errors encountered in the program. The message is listed at the end of every assembly on the terminal screen and in the listing file. When the message appears, the assembler has finished. When the message No Fatal Errors appears, the assembly is complete and successful.

Switches (/Switch)

You can command MACRO-80 to perform some additional functions besides assembling and creating object and listing files. These additional commands are given to MACRO-80 as entries at the end of the command line. A Switch entry directs MACRO-80 to "switch on" some additional or alternate function; hence, these entries are called switches. Switches are letters preceded by slash marks (/). Any number of switches may be entered, but each switch must be preceded by a slash. For example:

```
M80 ,=NEIL/L/R
```

The available switches for MACRO-80 are:

Switch	Action
/O	Octal listing. MACRO-80 generates List file addresses in octal radix.
/H	Hexadecimal listing. MACRO-80 generates List file addresses in hexadecimal. This is the default.
/R	Force generation of an Object file with the same name as the source file. May be used instead of giving a filename in the Object field of the command line. This switch is convenient when you want a REL file but forgot to enter a filename in the Object field and entered a comma and filename or a comma only in the List field. (Remember: if no filenames or comma is entered before the equal sign, a REL file is generated.) Thus, if you enter M80 ,NEIL=NEIL or M80 ,=NEIL then decide, before pressing <ENTER>, that you want a REL file, simply add /R. The command line would then be: M80 ,NEIL=NEIL/R or M80 ,=NEIL/R

/L Force generation of a listing file with the same name as the source file. May be used instead of giving a filename in the List field of the command line.

This switch is convenient when you want a List file but forgot to enter a filename in the List field. If you enter the command line:

```
    M80 =NEIL
 or M80 ,=NEIL
 or M80 NEIL=NEIL
```

then decide, before pressing <ENTER>, that you do want a List file, simply add /L. The command would then be:

```
    M80 =NEIL/L
 or M80 ,=NEIL/L
 or M80 NEIL=NEIL/L
```

/C Causes MACRO-80 to generate a special List file (with the same name as the Source file) for use with CREF-80 Cross Reference Facility. If you want to use CREF-80, you must assemble your file with this switch set. See Chapter 8, CREF-80 Cross Reference Facility, for further details.

/Z Directs MACRO-80 to assemble Z80 opcodes. If your source file contains Z80 opcodes and if you do not include the .Z80 pseudo-op in your source file, then you must use the /Z switch at assembly time so that MACRO-80 will assemble the Z80 opcodes.

/I Directs MACRO-80 to assemble 8080 opcodes. If your source file contains 8080 opcodes and if you do not include the .8080 pseudo-op in your source file, then you must use the /I switch at assembly time so that MACRO-80 will assemble the 8080 opcodes. (Default)

/P Each /P allocates an extra 256 bytes of stack space for use during assembly. Use /P if stack overflow errors occur during assembly. Otherwise, /P is not needed.

/M The /M switch initializes Block data areas. If you want the area that is defined by the DS (Define Space) pseudo-op initialized to zeros, then you should use the /M switch in the command line. Otherwise, the space is not guaranteed to contain zeros. That is, DS does not automatically initialize the space to zeros, in which case you may not know what is stored in the DS space or how the program will be affected.

/X The /X switch sets the default and current setting to suppress the listing of false conditionals. Absence of /X in the command line sets the default and current setting to list conditional blocks which evaluate false. /X is often used in conjunction with the conditional listing pseudo-op .TFCOND. Refer to the Listing Pseudo-ops section in Chapter 4 for details.

Additional Command Line Entries

Each command line field supports two additional types of entries--filename extensions and device designations. These two types of entries are actually part of a "file specification." A file specification includes the device where a file is located, the name of the file, and the filename extension.

Usually, filename extensions and device designations are handled by defaults--the MACRO-80 program "inserts" these entries if their positions are left blank in a command line. The default assignments in no way prevent you from entering either filename extensions or device designations, including entries that match the default entries. The programmer may enter or omit these additional entries in any combination.

The format for a file specification under MACRO-80 is:

```
dev:filename.ext
```

where: dev: is a 1-3 letter device designation followed by a (required) colon.

filename is a 1-8 letter filename.

.ext is a 1-3 character filename extension preceded by a (required) period.

Filename Extensions (.ext)

To distinguish between Source file, Object file, and List file, MACRO-80 appends an extension to each filename. Filename extensions are three-letter mnemonics appended to the filename with a period (.) between the filename and the extension. The extension which MACRO-80 appends reflects the type of file. Since the extensions are supplied by MACRO-80, they are called default extensions. The default extensions which MACRO-80 supplies are:

.REL	Relocatable object file
.PRN	Listing file
.COM	Absolute (executable object) file

Also, MACRO-80 assumes that, if no filename extension is entered, a source file carries the filename extension .MAC.

You may supply your own extensions, if you find this necessary or desirable. The disadvantage is that whenever you call the file, you must always remember to include your extension. Also, you must remember what type of file it is--relocatable, source, absolute, etc. The advantage of allowing MACRO-80 to assign default extensions is that you always have a mnemonic indication of the type of file, and you can call the filename without appending the extension, in most cases.

Device Designations (dev:)

Each of the fields in a command line (except Invocation) also may include a device designation.

When a device designation is specified in the Source field, the designation tells MACRO-80 where to find the source file. When a device designation is specified in the Object or List fields, the designation tells MACRO-80 where to output the object or list file. If the device designation is omitted from any of these fields, MACRO-80 assumes (defaults to) the currently logged drive. Thus, any time the device designation is the currently logged drive or device, the device designation need not be specified.

It is important to include device designations if several devices or drives will be used during an assembly. For example, if your ALDS diskette is in drive A and your program diskette is in drive B, and you want your REL file output to drive B, you need to give the command line:

```
M80 =B:NEIL
```

When the REL file is output, the currently logged drive is drive B. (However, when MACRO-80 is finished, drive A will be the currently logged drive again.) In contrast, if you saved your source program on the MACRO-80 diskette in drive A and want the REL file output to a diskette in drive B, then you need to enter the command line:

```
M80 B:=A:NEIL
```

As a rule of thumb, if you are not sure if you need to include the device designation (especially the drive designation), enter a designation; it is the one sure way to get the right files in the right places.

The available device designations for MACRO-80 are:

A:, B:, C:,...	Disk drives
LST:	Line Printer
TTY:	Terminal Screen or Keyboard
HSR:	High Speed Reader

Device Designations as Filenames

As an option, you may enter a device designation only in the command line fields in place of a filename. The use of this option gives various results depending on which device is specified and in which field the device is specified. For example:

```
M80 ,TTY:=TTY:
```

allows you to assemble a line immediately on input to check for syntax or other errors. A modification of this command (that is, M80 ,LST:=TTY:), provides the same result but printed on a line printer instead of the terminal screen.

If you use either of these commands (,TTY:=TTY: or ,LST:=TTY:), your first entry should be an END statement. You need to put the assembler into pass 2 before it will emit the code. If you simply start entering statement lines without first entering END, you will receive no response until an END statement is entered. Then you will have to reenter all your statements before you see any code generated.

The following table illustrates the results of the various choices. The table is meant to indicate the possibilities rather than provide an exhaustive list of the combinations.

dev:	Object	,List	=Source
A:, B:, C:, D:	write file to drive specified	write file to drive specified	N/A (a filename must be specified)
HSR:	N/A (input only)	N/A (input only)	reads source program from high-speed reader
LST:	N/A (unreadable file format)	writes listing to line printer	N/A (output only)
TTY:	N/A (unreadable file format)	"writes" listing to screen	"reads" source program from keyboard

Figure 5.1: Effects of Device Designations without Filenames

only when a .PAGE pseudo-op is encountered in the source file, or whenever the current page size has been filled.

SUBTTL text is the text supplied with the .SUBTTL pseudo-op, if .SUBTTL was included in the source file. If no .SUBTTL was given in the source file, this space is left blank.

A blank line follows the header data. The text of the listing file begins on the next line.

The format of a listing line is:

```
[error] ####m xx xxxxm[w]    text
```

where: error represents a one-letter error code. An error code is printed only if the line contains an error. Otherwise, the space is left blank.

represents the location counter. The number is a 4-digit hexadecimal number or a 6-digit octal number. The radix of the location counter number is determined by the use of the /O or /H switch in the MACRO-80 command line Switch field. If no radix switch was given, the default radix is hexadecimal (4-digit).

m represents the PC mode indicator character. The possible symbols are:

'	Code Relative
"	Data Relative
!	COMMON Relative
<space>	Absolute
*	External

xx and xxxx represent the assembled code. xx represents a one-byte value. One-byte values are always followed immediately by a space. xxxx represents a two-byte value, with the high-order byte printed first (this is the opposite of the order in which they are stored). Two-byte values are followed by one of the mode indicators discussed above (indicated by the second m).

[w] represents a line in the MACRO-80 file that came from another file through an INCLUDE pseudo-op; or a line that is part of an expansion (MACRO, REPT, IRP, IRPC). For lines from an INCLUDE statement, a C is printed following the assembled code; for lines in an expansion, a plus sign (+) is printed following the assembled code. Otherwise, this space is blank.

text represents the rest of the line, including labels, operations, arguments, and comments.

Symbol Table Format

The symbol table listing page follows the same header data format as the file line pages. However, instead of a page number, the symbol table page shows PAGE S.

Then, in a symbol table listing, all macro names in a program are listed alphabetically. Next, all symbols are listed, also alphabetically. A tab follows each symbol, then the value of the symbol is printed. Each symbol value is followed by one of the following characters:

I	PUBLIC symbol
U	Undefined symbol
C	COMMON block name. The value shown for the COMMON block name is its length in bytes in hexadecimal or octal radix.
*	External symbol
<space>	Absolute value

' Program relative value

" Data relative value

! COMMON relative value

5.4 ERROR CODES AND MESSAGES

Errors encountered during assembly cause MACRO-80 to return either an error code or an error message. Error codes are one-character flags printed in column one of the listing file. If a listing file is not being printed on the terminal screen, the lines containing errors will nevertheless be printed on the terminal screen. Error messages are printed at the end of the listing file, or, if the listing file is not being displayed on the terminal screen, any error messages will be displayed at the end of the error code lines.

ERROR CODE	MEANING
A	Argument error. The argument to a pseudo-op is not in correct format or is out of range.
C	Conditional nesting error. ELSE without IF, ENDIF without IF, two ELSEs for one IF, ENDC without COND.
D	Double defined symbol. Reference to a symbol which has more than one definition.
E	External error. Use of an External is illegal in the flagged context. For example, FOO SET NAME or LXI B,2-NAME.
M	Multiply defined symbol. The definition is for a symbol that already has a definition.
N	Number error. An error in a number, usually a bad digit. For example, 8Q.

O Bad opcode or objectionable syntax.
ENDM, LOCAL outside a block; SET, EQU, or MACRO without a name; bad syntax in an opcode; or bad syntax in an expression (for example, mismatched parentheses, quotes, consecutive operators).

P Phase error.
The value of a label or EQU name is different during pass 2 from its value during pass 1.

Q Questionable.
Usually, a line is not terminated properly. For example, MOV AX,BX,. This is a warning error.

R Relocation.
Illegal use of relocation in an expression, such as abs-rel. Data, code, and COMMON areas are relocatable.

U Undefined symbol.
A symbol referenced in an expression is not defined. For some pseudo-ops, a V error is printed for pass 1 then a U error for pass 2. Compare with V error code definition below.

V Value error.
On pass 1 a pseudo-op which must have its value known on pass 1 (for example, .RADIX, .PAGE, DS, IF, IFE) has a value which is undefined. If the symbol is defined later in the program, a U error will not appear on the pass 2 listing.

ERROR MESSAGES

%No END statement

No END statement: either it is missing or it is not parsed because it is in a false conditional, unterminated IRP/IRPC/REPT block, or terminated macro.

Unterminated conditional

At least one conditional is unterminated at the end of the file.

Unterminated REPT/IRP/IRPC/MACRO

At least one block is unterminated.

Contents

CHAPTER 6

LINK-80 LINKING LOADER

The .REL files which MACRO-80 creates are not executable. To make a REL file executable, you need to load and link the REL file with the LINK-80 linking loader. The result is an executable object file.

Loading means physically placing the file in memory and assigning absolute addresses to the code and data in place of the relative addresses assigned by the assembler. This is one of the required steps for converting a relocatable (REL) file into an executable (COM) file.

Linking means that each loaded file (or module) that directs program flow outside itself (by a CALL, an EXTERNAL symbol, or an Include) will be "linked" to the module that contains the corresponding code.

LINK-80 can also save the assembled-and-linked program as an executable object program on disk in a file with the extension .COM. Consequently, any time you wish to run your program, you need only insert the disk which contains your COM file into an appropriate disk drive and "call" your program -- a simple process of typing in the filename you used to save the program, followed by a carriage return.

6.1 INVOKING LINK-80

To invoke LINK-80, enter:

```
L80
```

The program file L80.COM will be loaded. LINK-80 will display an asterisk (*) to indicate that the linking loader is ready to accept a command. The REL file(s) you want link-loaded must be available in a disk drive. If you have only one drive, you will need to swap diskettes in the drive at each step of the link-loading process.

6.2 LINK-80 COMMANDS

LINK-80 commands are filenames and switches.

You can enter your commands to LINK-80 one at a time; or, you can enter all of your commands (including L80) on one line.

A command line has a flexible format, allowing you a number of options for loading and linking files and for performing other operations. The basic rule for LINK-80 commands is that files are loaded in the order they are named, beginning at the (default) address 103H under CP/M. Even though the files will be loaded in the order entered, you do not have to enter the files in the order of execution. LINK-80 places a jump instruction at address 100H-102H which jumps to the start address of the first instruction to be executed, regardless of its location in memory.

LINK-80 can perform about eleven different tasks. Even though you could use them all, you will rarely use more than three or four at a time.

When you enter a command to LINK-80, LINK-80 returns an asterisk (*) prompt that tells you to enter another command. For example:

```
A>L80<RETURN>
*/switch<RETURN>
*filename<RETURN>
*/switch<RETURN>
*filename/switch<RETURN>
*/E<RETURN>      (to exit LINK-80)
```

Note that all of the above lines may be entered as one line. For example:

```
L80 /switch,filename/switch,filename/switch/E<RETURN>
```

This shows further the flexibility of the LINK-80 command line.

Although entering each command on a separate line is slow and tedious, the advantage is, especially if you are new to a linking loader, that you know at all times what function LINK-80 is performing.

6.2.1 Filenames

Files processed by LINK-80 are REL files. A filename commands LINK-80 to load the named file (also called a module). If any file has been loaded already, a filename also commands LINK-80 to link the loaded files as required.

Normally each linking session requires at least two filenames. One filename directs LINK-80 which REL file to load and link; the other commands LINK-80 to save the executable code in a file with the specified name.

If you enter only one filename during the link session, either the COM file will not be saved (in which case you may have wasted your time), or LINK-80 will return the error message

```
?NOTHING LOADED
```

Note, however, that if you enter only one filename followed by the /G switch, the COM file will not be saved, but the program will execute as soon as LINK-80 is finished loading and linking. (Refer to the description of the switches in the next section.)

You may enter as many filenames as will fit on one line. The files named may be REL files in different languages (BASIC, COBOL, FORTRAN, or assembly) or runtime library REL files for any of the high-level programming languages. (For exact procedures for high-level language REL files, see the product manual included with the high-level language compiler.)

When LINK-80 is finished, the results are saved in the file named by the programmer in the command line (the filename followed by a /N -- see below, Section 6.2.2, Switches). LINK-80 gives this filename the extension .COM.

A filename command in LINK-80 actually means a file specification. A file specification includes a device designation, a filename, and a filename extension. The format of a file specification is:

```
dev:filename.ext
```

LINK-80 defaults the dev: to the default or currently logged disk drive. LINK-80 defaults the input filename extension to .REL and the output filename extension to .COM. You can alter the device designation to any applicable output device supported by MACRO-80 and/or the filename extension to any three characters by specifying a device or a filename extension when you enter a filename command.

6.2.2 Switches

Switches command LINK-80 to perform functions besides loading and linking. Switches are letters preceded by slash marks (/). You can place as many switches as you need in a single command line, but each switch letter must be preceded by a slash mark (/). For example, if you want to link and load a program named NEIL, save an image of it on diskette, then execute the program, you need two filenames and two switches, so you would enter the commands:

```
NEIL,NEIL/N/G<RETURN>
```

LINK-80 saves a memory image on diskette (the /N switch), then runs the NEIL program (the /G switch).

Some switches can be entered by themselves (/E, /G, /R, /P, /D, /U, /M, /O, /H). Some switches must be appended to the filename they affect (/N, /S). Some switches work only if other switches are also entered during the LINK-80 session (/X, /Y). Some switches must precede any filenames you want affected (/P, /D). Some switches command actions that are deferred until the end of the LINK-80 session (/N, /X, /Y). Some switches command actions that take place when entered (/S, /R -- a filename entered without a switch appended acts this way, too). These "rules of behavior" should be kept in mind when entering LINK-80 commands. See the descriptions for each switch for full details of its action.

The chart below summarizes the switches by function. Full descriptions of the switches by function follow the chart.

BE CAREFUL: Do not confuse the LINK-80 switches with the MACRO-80 switches.

FUNCTION	SWITCH	ACTION
Execute	/G	Execute .COM file then exit to operating system.
	/G:Name	Set .COM file start address equal to value of Name, execute .COM file, then exit to operating system.
Exit	/E	Exit to operating system.
	/E:Name	Set .COM file start address equal to value of Name, then exit to operating system.
Save	/N	Save all previously loaded programs and subroutines using filename immediately preceding /N.
	/N:P	Alternate form of /N; save only program area.
Address Setting	/P	Set start address for programs and data. If used with /D, /P sets only the program start.
	/D	Set start address for data area only.
	/R	Reset LINK-80.
Library Search	/S	Search the library named immediately preceding /S.
Global Listing	/U	List undefined globals.
	/M	List complete global reference map.
Radix Setting	/O	Octal radix.
	/H	Hexadecimal radix (default).
Special Code	/X	Save "COM" file in Intel ASCII Hex format. Requires /N switch. Gives "COM" file the extension .HEX.
	/Y	Creates a special file for use with SID/ZSID debugger. Requires /N and /E switches. Gives special file the extension .SYM.

Figure 6.1: Table of LINK-80 Switches

At least two switches will probably be used in every linking session. These switches belong to the first three functions -- Execute, Exit, and Save.

EXECUTE

Switch Action

/G The /G switch causes LINK-80 to load the filename(s) entered in the command line, to link the program(s) together, then to execute the link-loaded program. After the program run, your computer returns to operating system command level. For example,

```
L80 NEIL,NEIL/N/G
```

links NEIL.REL, saves the result in a disk file named NEIL.COM, then exits to the operating system.

Execution takes place as soon as the command line has been interpreted. Just before execution begins, LINK-80 prints three numbers and a BEGIN EXECUTION message. These three numbers can be very useful to you in developing future assembly language programs. The first number is the start address of the program. The second number is the address of the next available byte; that is, the end address plus one byte. The third number is the number of 256-byte pages taken up by the program (the difference between the start address and the end address converted to 256-byte pages).

If you do not want to save the .COM file, use the /G switch and enter only one filename on the command line. For example:

```
L80 NEIL/G
```

But Remember: No COM file is created (since you did not include /N). To run the program again, you will have to run LINK-80 again.

/G:<name> The /G:<name> switch performs exactly like the plain /G switch but with one additional feature. <name> is a global symbol which was defined previously in one of the modules which is being linked and loaded. When LINK-80 sees <name>, it uses <name> as the start of the program and loads the address of the line with <name> as its LABEL into the jump instruction at 100H-102H.

The value of this switch (and of /E:<name> below) is the ability to tell LINK-80 where to start execution when the assembled modules do not make this clear. Usually this is no problem because you link in a high-level language program (which LINK-80 takes as the main program by default), or you link only assembly language modules and only one has an END <name> statement to signal LINK-80 which assembly language program to execute first. But if two or more assembly language modules contain an END <name> statement, or if none of the assembly language modules contain an END <name> statement, then /G:<name> tells LINK-80 to use this module as the starting point for execution.

Programmers who want to execute an assembly language module before a high-level language program should use a CALL or INCLUDE statement at the beginning of the high-level language program to cause execution of the assembly language module before execution of the high-level language program.

EXIT

Switch	Action
/E	Use /E to link and load a program and perform some other functions on the files (for example, save it on a diskette) when you do not want to run the program at this time. When LINK-80 has finished the tasks, it will exit to the operating system. (The /G switch is the only other switch which exits LINK-80.) When linking is finished, LINK-80 outputs three numbers: start address, next available byte, number of 256-byte pages.
/E:<name>	The /E:<name> switch performs exactly like the plain /E switch but with one additional feature. <name> is a global symbol which was defined previously in one of the modules which is being linked and loaded. When LINK-80 sees <name>, it uses <name> as the start of the program and loads the address of the line with <name> as the LABEL into the jump instruction at 100H-102H. The value of this switch (and of /G:<name> above) is the ability to tell LINK-80 where to start execution when the assembled modules do not make this clear. Usually this is no problem because you link in a high-level language program (which LINK-80 takes as the main program by default), or you link only assembly language modules and only one has an END <name> statement to signal LINK-80 which assembly language program to execute first. But if two or more assembly language modules contain an END <name> statement, or if none of the assembly language modules contain an END <name> statement, then /E:<name> tells LINK-80 to use this module as the starting point for execution. Programmers who want to execute an assembly language module before a high-level language program should use a CALL or INCLUDE statement at the beginning of the high-level language program to cause this order of execution.

SAVE

Switch	Action
/N	The /N switch causes the assembled-linked program to be saved in a disk file. It is important that a filename always be specified for the /N switch. If you do not specify an extension, the default extension for the saved file is .COM. The COM filename will be the name the programmer appends the /N switch to. The /N switch must immediately follow the filename under which you wish to save the results of the link-load session.

The /N switch does not take effect unless a /E or /G switch follows it.

The most common error programmers make with the /N switch is to forget that they must specify at least two filenames; one as the file to be linked and another one as the name for the file to be saved. Therefore, at a minimum the command line should include:

```
L80 NEIL,NEIL/N/G
```

The first filename NEIL is the file to be loaded and linked; the second filename NEIL is the name for the COM file that will save the result of the link-loading session.

It is, of course, possible to specify filenames in any order. For example:

```
L80 NEIL/N,ASMSUB1,ASMSUB2,BASPROG/G
```

Here LINK-80 will load and link the files BASPROG, ASMSUB1, and ASMSUB2; then save the result in the file named NEIL.

From these two examples, it is possible to see that the filename followed by the /N save switch is not loaded; it is only a specification for an output file; you must also always name at least one input file, too.

You will use this switch almost every time you link a REL file because there is no other way to save the result of a link-load session and because not saving the result means you would have to link load again to run your program.

Once saved on disk, you need only type the COM filename at operating system command level to run the program.

/N:P By default, LINK-80 saves both the program and data areas in the COM file. If you wish to save only the program area to make your disk files smaller, use the /N switch in the form /N:P. With this switch set, only the program code will be saved.

Two of these switches (/N plus either a /G or a /E type) are all the switches required for most LINK-80 operations. Some additional functions are available through the use of other switches which allow programmers to manipulate the LINK-80 processes in more detail. The switches which turn on these additional functions are arranged in categories according to type of function. The function of each category is defined by the category name.

ADDRESS SETTING

Switch Action

/P The /P switch is used to set both the program and data origin. If you do not enter the /P switch, LINK-80 performs this task automatically, using a default address for both program and data. (103H for CP/M)

The format of the /P switch is:

/P:<address>,

The address value must be expressed in the current radix. The default radix is hexadecimal.

The /P switch is designed to allow you to place program (or code) segments at addresses other than the default. The default value for the /P switch is 103H.

REMEMBER: The /P switch takes effect as soon as it is seen, but it does not affect files already loaded. So be sure to place the /P switch before any files you want to load starting at the specified address. The /P switch and /D switch, when used, must be separated from the REL filename by a comma. For example,

L80 /P:103,NEIL,NEIL/N/E

The /P switch affects primarily the CSEG code in your assembly language program. If /P is given but not /D, both data and program (CSEG and DSEG) areas will be loaded starting at the /P:<address>. DSEG (and any COMMON areas) will be loaded first. If both /P and /D switches are given, /P sets the start of the CSEG area only. Normally, unless your programs are all CSEG, you will use /P and /D together.

Note especially that ASEG areas are not affected by the /P switch. So be careful to set the /P address outside any ASEG areas unless you want the program or data areas to write over the ASEG areas.

You may enter more than one /P switch during a single link session to place different program (code) segments at addresses which are not end to end. LINK-80 will automatically place one program segment (CSEG) after the next. You can cause space to be left between modules. However, some

restrictions one the placement of modules apply:

1. Be sure that program areas do not overlay one another. LINK-80 returns a warning error message if they do.

2. Be sure that the program areas are not split by data or COMMON areas; that is, a CSEG at 200H, a DSEG at 300H, and another CSEG at 400H is illegal. LINK-80 returns a fatal error in this case.

When the loading session is all done, LINK-80 wants to see a segment of memory loaded with data and COMMON and another segment loaded with program code. The code segments may have gaps between the modules as long as a data segment is not loaded between the start of the first code segment module and the end of the last code segment module, and vice versa. So, placing DSEG modules at 103H-115H, 150H-165H, 170H-175H, and CSEG modules at 200H-250H, 300H-350H, 400H-450H is acceptable. LINK and 80 will show Data between 103H and 175H and Program between 200H and 450H.

Note that any gaps you leave may contain data or program code from a previous program. LINK-80 does not initialize gaps to zero or null. This could cause unpredictable results.

/D The /D switch sets the origin for DSEG and COMMON areas. If you do not enter the /D switch, LINK-80 performs this task automatically, using a default address for both data and program. (103H for CP/M)

The format for the /D switch is:

```
/D:<address>,
```

The address for the /D switch must be in the current radix. The default radix is hexadecimal.

The /D switch is designed to allow you to place data and COMMON segments at addresses other than the default. The default value for the /D switch is 103H. The /D switch must be separated from the REL filenames by a comma. For example,

```
L80 /D:103,NEIL,NEIL/N/E
```

When the /P switch is used with the /D switch, data and common areas load starting at the address given with the /D switch. (The program will be

loaded beginning at the program origin given with the /P switch.) This is the only occasion when the address given in /P: is the start address for the actual program code.

REMEMBER: The /D switch takes effect as soon as LINK-80 "sees" the switch, so the /D switch has no effect on programs or data already loaded. Therefore, it is important to place the /D switch (as well as the /P switch) before the files you want to load starting at the address specified.

You may enter more than one /D switch during a single link session to place different program (code) segments at addresses which are not end to end. LINK-80 will automatically place one data segment (DSEG) after the next. You can cause space to be left between modules. However, some restrictions on the placement of modules apply:

1. Be sure that data areas do not overlay one another. LINK-80 returns a warning error message if they do.

2. Be sure that the data areas are not split by program areas; that is, a DSEG at 200H, a CSEG at 300H, and another DSEG at 400H is illegal. LINK-80 returns a fatal error in this case.

When the loading session is all done, LINK-80 wants to see a segment of memory loaded with data and COMMON and another segment loaded with program code. The data segments may have gaps between the modules as long as a program segment is not loaded between the start of the first data segment module and the end of the last data segment module, and vice versa. So, placing DSEG modules at 103H-115H, 150H-165H, 170H-175H, and CSEG modules at 200H-250H, 300H-350H, 400H-450H is acceptable. LINK and 80 will show Data between 103H and 175H and Program between 200H and 450H.

Note that any gaps you leave may contain data or program code from a previous program. LINK-80 does not initialize gaps to zero or null. This could cause unpredictable results.

ADDITIONAL NOTE FOR /P AND /D SWITCHES

If your program is too large for the loader, you will sometimes be able to load it anyway if you use /D and /P together. This way you will be able to load programs and data of a larger combined total. While LINK-80 is loading and linking, it builds a table consisting of five bytes for each program relative reference. By setting both /D and /P, you eliminate the need for LINK-80 to build this table, thus giving you some extra memory to work with.

To set the two switches, look to the end of the List file. Take the address you decided for the /D switch (where you want the DSEG to start loading), add the number for the total of data, add that number to 103H, add another 100H+1, and the result should be the /P: address for the start of the program area. The /D switch should be set at 103H or higher (D:103).

/R The /R switch "resets" LINK-80 to its initialized condition. LINK-80 scans the command line before it begins the functions commanded. As soon as LINK-80 sees the /R switch, all files loaded are ignored, LINK-80 resets itself, and the asterisk (*) prompt is returned showing that LINK-80 is running and waiting for you to enter a command line.

LIBRARY SEARCH

Switch	Action
/S	The /S switch causes LINK-80 to search the file named immediately prior to the switch for routines, subroutines, definitions for globals, and so on. In a command line, the filename with the /S switch appended must be separated from the rest of the command line by commas. For example:

```
L80 NEIL/N,MYLIB/S,NEIL/G
```

The /S switch is used to search library files only, including a library you constructed, using the LIB-80 Library Manager (see Chapter 8).

GLOBAL LISTING

Switch Action

/U The /U switch tells LINK-80 to list all undefined globals. The /U works only in command lines that do not include either a /G or a /E switch. Note that if your program contains any undefined globals, LINK-80 lists them automatically, unless the command line also contains a /S (library search) switch. In these cases, enter only the /U switch, and the list of undefined globals will be listed. Use CTRL-S to suspend the listing if you want to study a portion of the list that would scroll off the screen. Use CTRL-Q to restart the listing.

The various runtime libraries provide definitions for the globals you need to run your high-level language programs.

In addition to listing undefined globals, the /U switch directs LINK-80 to list the origin, end, and size of the program and data areas. These areas are listed as one lump area unless both the /P and /D switches are set. If both /P and /D are set, the start, end, and size of both areas are listed separately.

/M The /M switch directs LINK-80 to list all globals, both defined and undefined, on the screen. The listing cannot be sent to a printer. In the listing, defined globals are followed by their values, and undefined globals are followed by an asterisk (*).

In addition to listing all globals, the /M switch directs LINK-80 to list the origin, end, and size of the program and data areas. These areas are listed as one lump area unless both the /P and /D switches are set. If both /P and /D are set, the start, end, and size of both areas are listed separately.

RADIX SETTING

Switch	Action
/O	The /O switch sets the current radix to Octal. If you have a reason to use octal values in your program, give the /O switch in the command line. If you can think of no reason to switch to octal radix, then there is no reason to use this switch.
/H	The /H switch resets the current radix to Hexadecimal. Hexadecimal is the default radix. You do not need to give this switch in the command line unless you previously gave the /O switch and now want to return to hexadecimal.

SPECIAL CODE

Switch Action

/X The /X switch saves the "COM" file in Intel ASCII HEX format. The /X switch requires the /N switch appended to the same filename as the /X. For example:

```
L80 NEIL,NEIL/X/N/E
```

The file that is saved with the /X switch set is given the filename extension .HEX.

The primary use of the /X switch is to prepare programs to be burned into PROMs. The hex format was originally developed to facilitate the movement of programs from one machine to another. The hex format provides more code checking than object code does. Also, a HEX file can be edited with some sophisticated line editors.

/Y The /Y switch saves a file in a special format for use with Digital Research's Symbolic Debuggers, SID and ZSID. The /Y switch requires the /N and the /E (not /G) switches be given in the command line. For example:

```
L80 NEIL,NEIL/Y/N/E
```

The file that is saved with the /Y switch set is given the filename extension .SYM. A COM file will also be saved. So the sample command line above creates both NEIL.COM and NEIL.SYM.

The SYM file contains the names and addresses of all globals, which allows you to use Digital Research's Symbolic Debuggers SID and ZSID with the SYM file.

6.3 ERROR MESSAGES

Errors encountered during the running of LINK-80 will return messages, most preceded by either the symbol ? or the symbol %. No error codes are returned, so once you understand the meaning of the message, error recognition should be easy.

?No Start Address

The /G switch was issued, but no main program has been loaded.

?Loading Error

The last file given for input was not a properly formatted LINK-80 object file.

?Out of Memory

Not enough memory to load the module.

?Command Error

Unrecognizable LINK-80 command.

?<filename> Not Found

<filename>, as given in the command string, did not exist.

?Start Symbol - <name> - Undefined

The /E:Name or /G:Name switch was given, but the Name specified was not defined.

?Nothing Loaded

A <filename>/S or /E or /G was given, but no object file was loaded. That is, an attempt was made to search a library, to exit LINK-80, or to execute a program, when in fact nothing had been loaded. For example:

```
TEST/N/E
```

Results in "?Nothing Loaded" because TEST/N names TEST.COM, but does not load TEST.REL.

To load a file, enter the filename. To save a file, enter a filename followed by the /N switch and either a /E or a /G switch. For example, any of the following sets of commands should work:

- L80 NEIL,NEIL/N/E

  or

  ```
  L80
  *NEIL
  *NEIL/N/E
  ```

  or

  ```
  L80 NEIL/N,NEIL/E
  ```

?Can't Save Object File

A disk error occurred when the file was being saved. Usually, this means that the disk is full or that it is write-protected.

%2nd COMMON larger /XXXXXX/

When loading modules which include COMMON blocks, LINK-80 takes the size of the first COMMON block loaded to set the amount of memory needed before program code is loaded. If a subsequent module contains a COMMON block larger than the first one loaded, LINK-80 returns this error message. It means that the first definition of the COMMON block /XXXXXX/ encountered in the modules loaded was not the largest block defined with that name. Reorder module loading sequence or change COMMON block definitions so that all blocks are the same size.

%Mult. Def. Global YYYYYY

You have one global (PUBLIC) symbol name YYYYYY with more than one definition. Usually, two or more of the modules being loaded have declared the same symbol name as PUBLIC.

%Overlaying Program Area ,Start = xxxx
,Public = <symbol name> (xxxx)
,External = <symbol name> (xxxx)

Usually this occurs when either /D or /P is set to an address inside the area taken by LINK-80. You should reset the switch address above 102H. It may also occur if you set addresses for programs loaded after some initial programs were loaded and the addresses were not set high enough. For example, if MYPROG is larger than 147 bytes and you enter the commands:

```
MYPROG,/P:150,SUBR1,FUNNY/N/E
```

you will receive the %Overlaying Program Area error message.

%Overlaying Data Area ,Start = xxxx
,Public = <symbol name> (xxxx)
,External = <symbol name> (xxxx)

The /D and /P switches were set too close together. For example, if /D was given a higher address than /P but not high enough to be beyond the end of the program area, when the program is loaded, the top end will be laid over the data area. Or, if /D is lower than /P, /P was not high enough to prevent the beginning of the program from starting in the area already loaded with data.

?Intersecting Program Area
or
?Intersecting Data Area

The program and data areas intersect and an address or external chain entry is in this intersection. The final value cannot be converted to a current value since it is in the area intersection.

Origin Above Loader Memory, Move Anyway (Y or N)?
or
Origin Below Loader Memory, Move Anyway (Y or N)?

This message will appear only after either the /E or the /G switch command was given to LINK-80. If LINK-80 has not enough memory to load a module but a /E or /G has not been entered, you will receive the ?Out of Memory message.

LINK-80 can load modules only between its first address in memory and the top of available memory. If the program is too large for this space or if you set a /D and/or /P switch too high for the size of your program, LINK-80 runs out of memory and returns the Origin Above Loader Memory message.

If you set a /D and/or /P switch below the first address of LINK-80 (100H for CP/M), LINK-80 returns the Origin Below Loader Memory message. This prevents you from loading your program into memory designated for the operating system.

If a Y<CR> is given, LINK-80 will move the area and continue. If anything else is given, LINK-80 will exit. In either case, if the /N switch was given, the image will already have been saved.

Contents

CHAPTER 7

CREF-80 CROSS REFERENCE FACILITY

A cross reference facility processes a specially assembled listing file to list the locations of all intermodule references and the locations of their definitions. The result is a cross reference listing. This cross reference listing can be used to aid debugging your program.

The CREF-80 Cross Reference Facility allows a programmer to process the cross reference file generated by MACRO-80. This cross reference file contains embedded control characters, set up during MACRO-80 assembly. CREF-80 interprets the control characters and generates a file that lists cross references among variables.

CREF-80 produces a listing, resembling the PRN listing of MACRO-80, with two additional features:

1. Each source statement is numbered with a cross reference number.

2. At the end of the listing, variable names appear in alphabetic order. Each name is followed by the line number where the variable is defined (flagged with #) followed by the numbers of other lines where the variable is referenced.

The CREF listing file replaces the MACRO-80 PRN List file and receives the filename extension .LST instead of .PRN.

7.1 CREATING A CREF LISTING

Creating a CREF listing involves two steps: (1) creating a cross reference file (.CRF), and (2) generating a cross reference listing (.LST). The first step occurs in the MACRO-80 macro assembler; the second in the CREF-80 Cross Reference Facility.

Creating a Cross Reference File

To create a cross reference file, set the /C switch in the MACRO-80 command line. For example:

```
M80 =NEIL/C
```

This command line assembles the file NEIL.MAC, generating the output files NEIL.REL (object file) and NEIL.CRF (cross reference file).

Generating a Cross Reference Listing

The cross reference listing is generated by running the .CRF file through CREF-80.

To invoke the cross reference facility, enter:

```
CREF80
```

CREF-80 will return an asterisk (*) prompt.

To create the cross reference listing file, enter:

```
=filename
```

where filename is the name of your .CRF file. For example:

```
CREF80 =NEIL
```

will generate a .LST file (NEIL.LST) containing the cross reference information.

This .LST file can be printed or sent to the terminal screen using operating system commands. Additionally, CREF-80 supports the same output device designations as MACRO-80. Simply enter the device designation in front of the filename. For example:

```
CREF80 LST:=NEIL
```

sends the .LST listing to the printer only (no disk file is generated).

```
CREF80 TTY:=NEIL
```

sends the .LST listing to the CRT only (no disk file is generated).

You will need to give a drive designation if you want the .LST file saved elsewhere than the currently logged drive (where the .CRF file resides). For example:

```
CREF80 B:=A:NEIL
```

saves NEIL.LST on drive B.

When finished, CREF-80 prompts with an asterisk. You may enter another =filename, or exit from CREF-80 to the operating system.

To exit CREF-80, enter:

```
CTRL-C
```

If you want the .LST file named differently from the default (.CRF filename and extension .LST), enter the name in front of the equal sign. For example:

```
CREF80 NEIL.CRL=NEIL
or CREF80 NEILCREF=NEIL
```

The former command line generates a cross reference list file named NEIL.CRL; the latter generates a file named NEILCREF.LST.

Look at the filename extensions to distinguish a cross reference listing file from the listing file MACRO-80 normally generates. The listing file MACRO-80 normally generates (without the /C switch set in the command line) receives the default filename extension .PRN. The cross reference listing file generated by CREF-80 receives the default filename extension .LST.

7.2 CREF LISTING CONTROL PSEUDO-OPS

You may want the option of generating a cross reference listing for part of a program but not all of it. To control the listing or suppressing of cross references, use the cross reference listing control pseudo-ops, .CREF and .XCREF, in the source file for MACRO-80. These two pseudo-ops may be entered at any point in the program in the OPERATOR field. Like the other listing control pseudo-ops, .CREF and .XCREF support no ARGUMENTs.

Pseudo-op	Definition
.CREF	Create cross references. .CREF is the default condition. Use .CREF to restart the creation of a cross reference file after using the .XCREF pseudo-op. .CREF remains in effect until MACRO-80 encounters .XCREF. Note, however, that .CREF has no effect until the /C switch is set in the MACRO-80 command line.
.XCREF	Suppress cross references. .XCREF turns off the .CREF (default) pseudo-op. .XCREF remains in effect until MACRO-80 encounters .CREF. Use .XCREF to suppress the creation of cross references in selected portions of the file. Because neither .CREF nor .XCREF takes effect until the /C switch is set in the MACRO-80 command line, there is no need to use .XCREF if you want the usual List file (one without cross references); simply omit /C from the MACRO-80 command line.

Contents

CHAPTER 8

LIB-80 LIBRARY MANAGER

WARNING

> Read this chapter carefully and make a back-up copy of your libraries before using LIB-80. LIB-80 is very powerful and thus can be very destructive. It is easy to destroy a library with LIB-80.

LIB-80 is designed as a runtime library manager for CP/M versions of Microsoft FORTRAN-80 and COBOL-80. LIB-80 may also be used to create your own library of assembly language subroutines.

LIB-80 creates runtime libraries from assembly language programs that are subroutines to COBOL, FORTRAN, and other assembly language programs. The programs collected by LIB-80 may be special modules created by the programmer or modules from an existing library (FORLIB, for example). With LIB-80, you can build specialized runtime libraries for whatever execution requirements you design.

The value of building a library is that all the routines needed to execute a program can be linked with it into an executable object (COM) file by entering the library name followed by /S in a LINK-80 command line. For example:

```
L80 MAIN,NEWLIB/S,NEIL/N/G
```

This is much more convenient than entering the necessary subroutines individually, especially if there are many modules. With a library file you can be sure all the necessary modules will be linked into the COM file, plus there is no danger of running out of space on the LINK-80

command line. Additionally, the library makes this special collection of subroutines available for easy linking into any program.

8.1 SAMPLE LIB-80 SESSION

The two most common uses you will have for LIB-80 are building a library and listing a library. The following sample sessions illustrate the basic commands for these two uses.

BUILDING A LIBRARY:

```
A>LIB
*TRANLIB=SIN,COS,TAN,ATAN,ACOG
*EXP
*/E
A>
```

In this sample session, LIB invokes LIB-80, which returns an asterisk (*) prompt. TRANLIB is the name of the library being created. SIN,COS,TAN,ATAN,ACOG are filenames to be concatenated into TRANLIB. EXP is another filename to be concatenated into TRANLIB. (EXP could be listed on the previous command line; this example shows files entered singly and multiply.) /E causes LIB-80 to rename TRANLIB.LIB to TRANLIB.REL then to exit to CP/M.

LISTING A LIBRARY:

```
A>LIB
*TRANLIB.LIB/U
*TRANLIB.LIB/L
    .
    .
    .

(List of symbols in TRANLIB.LIB)
    .
    .
    .

*CTRL-C
A>
```

In this sample session, LIB invokes LIB-80. TRANLIB.LIB/U tells LIB-80 to search TRANLIB.LIB for any intermodule references that would not be defined during a single pass through the library

(that is, any "backward" referencing symbols). TRANLIB.LIB/L directs LIB-80 to list the modules in TRANLIB.LIB and the symbol definitions the modules contain. CTRL-C exits to CP/M without destroying any files.

WARNING

/E will destroy your current library if there is no new library under construction. This is a special danger to your FORTRAN runtime library FORLIB.REL. IF YOU ARE ONLY LISTING THE LIBRARY AND NOT REVISING IT, EXIT LIB-80 USING CTRL-C.

8.2 LIB-80 COMMANDS

Invoking LIB-80

To invoke LIB-80, enter:

```
LIB
```

LIB-80 will return an asterisk (*) prompt, indicating ready to accept commands. Each command in LIB-80 adds modules to the library under construction.

Commands to LIB-80 consist of an optional Destination field, a Source field, and an optional Switch field.

The format of a LIB-80 command is:

```
Destination=Source/Switch
```

Each field is described below. The general format for each field is shown in parentheses after the field name.

Destination field (filename=)

This field is optional. The equal sign is required if any entry is made in this field.

Enter in this field the filename (and extension, if you choose) for the library file you want to create.

If this field is omitted, LIB-80 defaults to the filename FORLIB. The default filename extension is .REL.

WARNING

Do not confuse this default filename FORLIB.LIB with FORLIB.REL, the runtime library supplied with FORTRAN-80. These two libraries will not be the same unless you command LIB-80 to copy all the files from the FORTRAN runtime library to the new library. Furthermore, when you exit LIB-80, the default library name will be given the filename extension .REL, which means that it replaces the FORLIB.REL supplied with FORTRAN-80. For this reason, unless you want your FORTRAN-80 runtime library destroyed, we recommend emphatically that you always specify a Destination filename when creating a new library.

Source field (filename<module>)

Some entry is required in this field. All Source files must be REL files.

Source field entries tell LIB-80 which files or parts of files (modules) you want added to the destination library file. You have two choices for entries:

1. Filename(s) only

2. Any combination of filename(s) and module name(s)

The following syntax rules apply:

1. If a command consists of filenames only, the entries are separated by commas only. For example:

   FILE1,FILE2,FILE3

2. If a command consists of filenames and module names, the module names must be enclosed in angle brackets (<>). Modules follow the filename where they are found. Each filename<module name> combination is separated from other command line entries by commas. For example:

   FILE1,FILE2<MODZ>,FILE3<MODR>,FILE4

3. If more than one module is named from the same file, the module names, enclosed in angle brackets (<>), must be separated from each other by commas. For example:

   FILE1,FILE2<MODZ,MODR>,FILE3

   See Additional Details about Source Modules, option 2, below.

Files and modules are typically FORTRAN or COBOL subprograms or main programs, or ALDS assembly language programs that contain ENTRY, GLOBAL, or PUBLIC statements. (These statements are called entry points.) LIB-80 recognizes a module by its program name, which may be a filename, or a name given by either the .TITLE or the NAME pseudo-op in MACRO-80. All Source files must be REL files.

LIB-80 concatenates REL files and modules of REL files; that is, LIB-80 strings one file or module after the other.

So there is no difference between the command under syntax rule 2 above and

```
FILE1
FILE2<MODZ>
FILE3<MODR>
FILE4
```

Also, because the library file is built by concatenation, it is important to order the modules so that all intermodule references are "forward." That is, the module containing the external reference should physically appear ahead of the module containing the ENTRY point (the definition). Otherwise, when you direct LINK-80 to search the library, LINK-80 may not satisfy all references on a single pass through the library.

Additional Details about Source Modules

To extract modules from previous libraries and other REL files, LIB-80 uses a powerful syntax to specify ranges of modules within a REL file.

These ranges may be from one module to the entire file (in which case no module specification is given).

The basic principle of specifying a range of modules is, generally, that any module named in a command will be included. (There is an exception, when specifying a relative offset range--item 6, below.)

The options for specifying modules are:

1. One module only
   Enter the module name. For example:

   ```
   FILE1<MODZ>
   ```

   includes only module MODZ of FILE1.

2. Several discontiguous modules from one file
   Enter the module names separated by commas. For example:

   ```
   FILE1<MODZ,MODR,MODK>
   ```

   includes modules MODZ, MODR, and MODK. Note that these modules may be given in any order you need them concantenated for a proper one-pass search, regardless of their order in the original file.

3. From the first module through the named module
   Enter two periods (..) and the name of the last module to be included. For example:

   FILE1<..MODK>

   includes all modules from the first module in FILE1 through module MODK.

4. From a named module through the last module
   Enter the name of the module that starts the range followed by two periods (..). For example:

   FILE1<MODR..>

   includes all the modules, beginning with module MODR, through the last module in FILE1.

5. From one named module through another named module
   Enter the name of the module that starts the range followed by two periods (..) followed by the name of the module that ends the range. For example:

   FILE1<MODZ..MODK>

   includes all modules, beginning with module MODZ, through module MODK.

6. Relative offset range
   Enter the module name followed by a + or - and the number of modules to be included. + means following the named module. - means preceding the named module. The named module is not included in the library. The offset number must be an integer in the range 1 to 255. For example:

   FILE1<MODZ+2>

   includes the two modules immediately following module MODZ. While

   FILE1<MODK-3>

   includes the three modules immediately preceding module MODK.

Additionally, ranges and offsets may be used together. For example:

```
FILE1<MODR+1..MODK-1>
```

includes all the modules between module MODR and module MODK (but neither MODR nor MODK is included).

7. All modules in a file
   Enter the filename only. For example:

```
FILE1
```

includes the entire file (all modules in FILE1).

Switch field (/switch)

An entry in the Switch field commands LIB-80 to perform additional functions. A Switch field entry is a letter preceded by a slash mark (/).

WARNING

/E will destroy your current library if there is no new library under construction. This is a special danger to your FORTRAN runtime library FORLIB.REL because FORLIB is the default filename used if you do not specify a destination filename. Therefore, unless you want to delete your complete FORTRAN runtime library, give LIB-80 a destination filename for the new library. If you are only listing the library and not revising it, exit LIB-80 using CTRL-C.

/O Use /O to set typeout mode to Octal radix. /O will be given together with the /L switch, which commands LIB-80 to list. REMEMBER: When switches are given together, a slash must precede each switch. For example:

```
NEWLIB/L/O
```

/H Use /H to set typeout mode to Hexadecimal radix. Hexadecimal is the default radix.

Contents

APPENDIX A

Compatibility with Other Assemblers

The $EJECT and $TITLE controls are provided for compatability with Intel's ISIS assembler. The dollar sign must appear in column 1 only if spaces or tabs separate the dollar sign from the control word. The control word

```
$EJECT
```

is the same as the MACRO-80 PAGE pseudo-op.

The control word

```
$TITLE('text')
```

is the same as the MACRO-80 SUBTTL <text> pseudo-op.

The Intel operands PAGE and INPAGE generate Q errors when used with the MACRO-80 CSEG or DSEG pseudo-ops. These errors are warnings; the assembler ignores the operands.

When MACRO-80 is invoked, the default for the origin is Code Relative 0. With the Intel ISIS assembler, the default is Absolute 0.

With MACRO-80, the dollar sign ($) is a defined constant that indicates the value of the location counter at the start of the statement. Other assemblers may use a decimal point or an asterisk. Other constants are defined by MACRO-80 to have the following values:

```
A=7     B=0     C=1     D=2     E=3
H=4     L=5     M=6     SP=6    PSW=6
```

APPENDIX B

The Utility Software Package with TEKDOS

The command formats for MACRO-80, LINK-80, and CREF-80 differ slightly under the TEKDOS operating system.

B.1 TEKDOS COMMAND FILES

The files M80, L80, and C80 are actually TEKDOS command files for the assembler, loader, and cross reference programs, respectively. These command files set the emulation mode to 0 and select the Z-80 assembler processor (see TEKDOS documentation), then execute the appropriate program file. You will note that all of these command files are set up to execute the Microsoft programs from drive #1. LINK-80 will also look for the library on drive #1. If you wish to execute any of this software from drive #0, the command file must be edited. Then, LINK-80 should be given an explicit library search directive, such as MYLIB-S. See the Switches section in Chapter 6, LINK-80 Linking Loader.

Filenames under TEKDOS do not use the Utility Software Package default filename extensions.

B.2 MACRO-80

The MACRO-80 assembler accepts command lines only (the invoke command, M80, and all filenames and switches must be on one line). No prompt is displayed, and the interactive commands (,TTY:=TTY: and ,LPT:=TTY:) are not accepted. Commands have the same format as TEKDOS assembler commands; that is, up to three filenames or device names plus optional switches.

M80 [object] [list] source [switch [switch [...]]]

The object and list file entries are optional. These files will not be created if the parameters are omitted. Any

error messages will still be displayed on the console. The available switches are described in Chapter 5 of this manual. All command line entries may be delimited by commas or spaces. If you do not want to request an object file, you must enter a <space comma space> between the M80 entry and the name of the list file. For example:

```
M80 , LIST SOURCE
```

B.3 CREF-80

The form of commands to CREF-80 is:

```
C80 list source
```

Both filenames are required. The source file is always the name of a CREF-80 file created during assembly by the C switch.

Example:

To create a CREF-80 file from the source TSTMAC using MACRO-80, enter:

```
M80 , TSTCRF TSTMAC C
```

To create a cross reference listing from the CREF-80 file TSTCRF, enter:

```
C80 TSTLST TSTCRF
```

B.4 LINK-80

With TEKDOS, the LINK-80 loader accepts interactive commands only. Command lines are not supported.

When LINK-80 is invoked, and whenever it is waiting for input, it will prompt with an asterisk. Commands are lists of filenames and/or devices separated by commas or spaces and optionally interspersed with switches. The input to LINK-80 must be Microsoft relocatable object code (not the same as TEKDOS loader format).

Switches to LINK-80 are delimited by hyphens under TEKDOS, instead of slashes. All LINK-80 switches (as documented in Chapter 6) are supported, except -G and -N, which are not implemented at this time.

EXAMPLE:

1. Assemble a MACRO-80 program named XTEST, creating an object file called XREL and a listing file called XLST:

```
>M80 XREL XLST XTEST
```

2. Load XTEST and save the loaded module:

```
>L80
*XREL-E
[04AD  22B8]
*DOS*ERROR 46
L80 TERMINATED
>M XMOD 400 22B8 04AD
```

Note that -E exits via an error message due to execution of a Halt instruction. The memory image is intact, however, and the TEKDOS Module command may be used to save it. Once a program is saved in module format, it may then be executed directly without going through LINK-80 again.

The bracketed numbers printed by LINK-80 before exiting are the entry point address and the highest address loaded, respectively. The loader default is to begin loading at 400H. However, the loader also places a jump to the start address in location 0, which allows execution to begin at 0. The memory locations between 0003 and 0400H are reserved for SRB's and I/O buffers at runtime.

APPENDIX C

ASCII CHARACTER CODES

Dec	Hex	CHR	Dec	Hex	CHR	Dec	Hex	CHR
000	00H	NUL	043	2BH	+	086	56H	V
001	01H	SOH	044	2CH	,	087	57H	W
002	02H	STX	045	2DH	-	088	58H	X
003	03H	ETX	046	2EH	.	089	59H	Y
004	04H	EOT	047	2FH	/	090	5AH	Z
005	05H	ENQ	048	30H	0	091	5BH	[
006	06H	ACK	049	31H	1	092	5CH	\
007	07H	BEL	050	32H	2	093	5DH	]
008	08H	BS	051	33H	3	094	5EH	^
009	09H	HT	052	34H	4	095	5FH	
010	0AH	LF	053	35H	5	096	60H	`
011	0BH	VT	054	36H	6	097	61H	a
012	0CH	FF	055	37H	7	098	62H	b
013	0DH	CR	056	38H	8	099	63H	c
014	0EH	SO	057	39H	9	100	64H	d
015	0FH	SI	058	3AH	:	101	65H	e
016	10H	DLE	059	3BH	;	102	66H	f
017	11H	DC1	060	3CH	<	103	67H	g
018	12H	DC2	061	3DH	=	104	68H	h
019	13H	DC3	062	3EH	>	105	69H	i
020	14H	DC4	063	3FH	?	106	6AH	j
021	15H	NAK	064	40H	@	107	6BH	k
022	16H	SYN	065	41H	A	108	6CH	l
023	17H	ETB	066	42H	B	109	6DH	m
024	18H	CAN	067	43H	C	110	6EH	n
025	19H	EM	068	44H	D	111	6FH	o
026	1AH	SUB	069	45H	E	112	70H	p
027	1BH	ESCAPE	070	46H	F	113	71H	q
028	1CH	FS	071	47H	G	114	72H	r
029	1DH	GS	072	48H	H	115	73H	s
030	1EH	RS	073	49H	I	116	74H	t
031	1FH	US	074	4AH	J	117	75H	u
032	20H	SPACE	075	4BH	K	118	76H	v
033	21H	!	076	4CH	L	119	77H	w
034	22H	"	077	4DH	M	120	78H	x
035	23H	#	078	4EH	N	121	79H	y
036	24H	$	079	4FH	O	122	7AH	z
037	25H	%	080	50H	P	123	7BH	{
038	26H	&	081	51H	Q	124	7CH	|
039	27H	'	082	52H	R	125	7DH	}
040	28H	(	083	53H	S	126	7EH	~
041	29H	)	084	54H	T	127	7FH	DEL
042	2AH	*	085	55H	U			

Dec=decimal, Hex=hexadecimal (H), CHR=character.
LF=Line Feed, FF=Form Feed, CR=Carriage Return, DEL=Rubout

APPENDIX D

FORMAT OF LINK COMPATIBLE OBJECT FILES

This appendix contains reference material for users who wish to know the load format of LINK-80 relocatable object files. None of this material is necessary to the operation of ALDS. There is nothing in the format material presented here which can be manipulated by the user. The material is highly technical, and it is not presented in any tutorial manner.

LINK-compatible object files consist of a bit stream. Individual fields within the bit stream are not aligned on byte boundaries, except as noted below. Use of a bit stream for relocatable object files keeps the size of object files to a minimum, thereby decreasing the number of disk reads/writes.

There are two basic types of load items: Absolute and Relocatable. The first bit of an item indicates one of these two types. If the first bit is a 0, the following 8 bits are loaded as an absolute byte. If the first bit is a 1, the next 2 bits are used to indicate one of four types of relocatable items:

00	Special LINK item (see below).
01	Program Relative. Load the following 16 bits after adding the current Program base.
10	Data Relative. Load the following 16 bits after adding the current Data base.
11	Common Relative. Load the following 16 bits after adding the current Common base.

Special LINK items consist of the bit stream 100 (read one-zero-zero) followed by:

a four-bit control field

an optional A field consisting of a two-bit address type that is the same as the two-bit field described above, except 00 specifies absolute address

an optional B field consisting of 3 bits that give a symbol length and up to 8 bits for each character of the symbol

A general representation of a special LINK item is:

```
1 00 xxxx    yy nn     zzz + characters of symbol name
             A field              B field
```

where:
- xxxx is four-bit control field (0-15 below)
- yy is two-bit address type field
- nn is sixteen-bit value
- zzz is three-bit symbol length field

The following special types have a B-field only:

0 Entry symbol (name for search)
1 Select COMMON block
2 Program name
3 Request library search
4 Extension LINK items (see below)

The following special LINK items have both an A field and a B field:

5 Define COMMON size
6 Chain external (A is head of address chain, B is name of external symbol)
7 Define entry point (A is address, B is name)

The following special LINK items have an A field only:

8	External - offset. Used for JMP and CALL to externals
9	External + offset. The A value will be added to the two bytes starting at the current location counter immediately before execution.
10	Define size of Data area (A is size)
11	Set loading location counter to A
12	Chain address. A is head of chain. Replace all entries in chain with current location counter. The last entry in the chain has an address field of absolute zero.
13	Define program size (A is size)
14	End program (forces to byte boundary)

The following special LINK item has neither an A nor a B field:

15	End file

An Extension LINK item follows the general format of a B-field-only special LINK item, but the contents of the B-field are not a symbol name. Instead, the symbol area contains one character to identify the type of extension LINK item, followed by from 1 to 7 characters of additional information.

Thus, every extension LINK item has the format:

1 00 0100 111 s bbbbbb

where:	111	is 3 bits containing the length of the field bbbbbb (0 implys 1 since F80 emits entry length of 0 for Blank Common),
	s	is an eight bit extension LINK item sub-type identifier, and
	bbbbbb	are 1 to 6 bytes for additional information. If used as B field for name, bbbbbb may be only 6 characters.

The present extension LINK item sub-types are:

5	X'35' COBOL overlay segment sentinel
A	X'41' Arithmetic Fixup (Arithmetic Operator)
B	X'42' Arithmetic Fixup (External Reference)
C	X'43' Arithmetic Fixup (Area Base + Offset)

Descriptions of Sub-types

Sub-type 5

When the overlay segment sentinel is encountered by LINK-80, lll receives the value 010 (binary), and the current overlay segment number is set to the value b+49. If the previously existing segment number was non-zero and the /N switch is in effect, the data area is written to disk in a file whose name is the current program name and whose extension is Vnn, where nn are the two hexadecimal digits representing the number b+49 (decimal).

Sub-types A,B,C

Sub-types A, B, and C allow the processing of Polish Arithmetic text. Items must be read as Reverse Polish Expression. One or more Value items (sub-type B or C) are followed by one or more Arithmetic Operators (sub-type A) and end with a Store-Result Arithmetic Operator (B.STBT or B.STWD).

All Items are put in the Fixup Table afer any offset entries have been converted to final addresses. The Polish expression is executed out of the Fixup Table at the end of link. The result is stored at the PC given when the Items were read.

APPENDIX E

Table of MACRO-80 Pseudo-ops

Notation: * means Z80 pseudo-op
no stars means 8080 pseudo-op

SINGLE-FUNCTION PSEUDO-OPS

Instruction Set Selection

```
        .Z80
        .8080
```

Data Definition and Symbol Definition

```
        <name> ASET <exp>
        BYTE EXT <symbol>
        BYTE EXTRN <symbol>
        BYTE EXTERNAL <symbol>
        DB <exp>[,<exp>...]
        DB <string>[<string>...]
        DC <string>
        DDB <exp>[,<exp>...]
    *   DEFB <exp>[,<exp>...]
    *   <name> DEFL <exp>
    *   DEFM <string>[,<string>...]
    *   DEFS <exp>[,<val>]
    *   DEFW <exp>[,<exp>...]
        DS  <exp>[,<val>]
        DW <exp>[,<exp>...]
        ENTRY <name>[,<name>...]
        <name> EQU <exp>
        EXT <name>[,<name>...]
        EXTRN <name>[,<name>...]
    *   EXTERNAL <name>[,<name>...]
        GLOBAL <name>[,<name>...]
        PUBLIC <name>[,<name>...]
        <name> SET <exp> (not in .Z80 mode)
```

PC Mode Pseudo-ops

```
ASEG
CSEG
DSEG
COMMON /<block name>/
ORG <exp>
.PHASE <exp>/.DEPHASE
```

File Related Pseudo-ops

```
.COMMENT <delim><text><delim>
END [<exp>]
INCLUDE <filename>
$INCLUDE <filename>
MACLIB <filename>
.RADIX <exp>
.REQUEST <filename>[,<filename>...]
```

Listing Pseudo-ops

Format Control Pseudo-ops

```
*  *EJECT [<exp>] (one star is part of *EJECT)
   PAGE <exp>
   SUBTTL <text>
   TITLE <text>
   $TITLE
```

General Listing Control Pseudo-ops

```
.LIST
.XLIST
.PRINTX <delim><text><delim>
```

Conditional Listing Control Pseudo-ops

```
.SFCOND
.LFCOND
.TFCOND
```

Expansion Listing Control Pseudo-ops

```
.LALL
.SALL
.XALL
```

Cross-Reference Listing Control Pseudo-ops

```
.XCREF
.CREF
```

MACRO FACILITY PSEUDO-OPS

Macro Pseudo-ops

```
<name> MACRO <parameter>[,<parameter>...]
ENDM
EXITM
LOCAL <parameter>[,<parameter>...]
```

Repeat Pseudo-ops

```
REPT <exp>
IRP <dummy>,<parameters in angle brackets>
IRPC <dummy>,string
```

Conditional Assembly Facility

```
*  COND <exp>
   ELSE
*  ENDC
   ENDIF
   IF <exp>
   IFB <arg>
   IFDEF <symbol>
   IFDIF <arg1>,<arg2>
   IFE <exp>
   IFF <exp>
   IFIDN <arg1>,<arg2>
   IFNB <arg>
   IFNDEF <symbol>
   IFT <exp>
   IF1
   IF2
```

APPENDIX F

Table of Opcodes

The opcodes are listed alphabetically by instruction set. For details, refer to the reference books listed in Chapter 1.

F.1 Z80 OPCODES

Opcode	Function
ADC A	Add with Carry to Accumulator
ADC HL,rp	Add Register Pair with Carry to HL
ADD	Add
AND	Logical AND
BIT	Test Bit
CALL addr	Call Subroutine
CALL cond,addr	Call Conditional
CCF	Complement Carry Flag
CP	Compare
CPD	Compare, Decrement
CPDR	Compare, Decrement, Repeat
CPI	Compare, Increment
CPIR	Compare, Increment, Repeat
CPL	Complement Accumulator
DAA	Decimal Adjust Accumulator
DEC	Decrement
DI	Disable Interrupts
DJNZ	Decrement and Jump if Not Zero
EI	Enable Interrupts
EX	Exchange
EXX	Exchange Register Pairs and Alternatives
HALT	Halt
IM x	Set Interrupt Mode
IN	Input
INC	Increment
IND	Input, Decrement
INDR	Input, Decrement, Repeat
INI	Input, Increment
INIR	Input, Increment, Repeat
JP addr	Jump
JP cond,addr	Jump Conditional
JR	Jump Relative

```
JR     cond,addr      Jump Relative Conditional
LD     A,(addr)       Load Accumulator Direct
LD   A,(BC) or (DE)   Load Accumulator Secondary
LD     A,I            Load Accumulator from Interrupt Vector Register
LD     A,R            Load Accumulator from Refresh Register
LD   HL,(addr)        Load HL Direct
LD     data           Load Immediate
LD   xy,(addr)        Load Index Register Direct
LD     reg,(HL)       Load Register
LD   reg,(xy+disp)    Load Register Indexed
LD   rp,(addr)        Load Register Pair Direct
LD     SP,HL          Move HL to Stack Pointer
LD     SP,xy          Move Index Register to Stack Pointer
LD     dst,scr        Move Register-to-Register
LD     (addr),A       Store Accumulator Direct
LD   (BC) or (DE),A   Store Accumulator Secondary
LD     I,A            Store Accumulator to Interrupt Vector Register
LD     R,A            Store Accumulator to Refresh Register
LD   (addr),HL        Store HL Direct
LD   (HL),data        Store Immediate to Memory
LD   (xy+disp),data   Store Immediate to Memory Indexed
LD   (addr),xy        Store Index Register Direct
LD     (HL),reg       Store Register
LD   (xy+disp),reg    Store Register Indexed
LD   (addr),rp        Store Register Pair Direct
LDD                   Load, Decrement
LDDR                  Load, Decrement, Repeat
LDI                   Load, Increment
LDIR                  Load, Increment, Repeat
NEG                   Negate (Two's Complement) Accumulator
NOP                   No Operation
OR                    Logical OR
OUT                   Output
OUTD                  Output, Decrement
OTDR                  Output, Decrement, Repeat
OUTI                  Output, Increment
OTIR                  Output, Increment, Repeat
POP                   Pop from Stack
PUSH                  Push to Stack
RES                   Reset Bit
RET                   Return from Subroutine
RET    cond           Return Conditional
RETI                  Return from Interrupt
RETN                  Return from Non-Maskable Interrupt
RL                    Rotate Left Through Carry
RLA                   Rotate Accumulator Left Through Carry
RLC                   Rotate Left Circular
RLCA                  Rotate Accumulator Left Circular
RLD                   Rotate Accumulator and Memory Left Decimal
RR                    Rotate Right Through Carry
RRA                   Rotate Accumulator Right Through Carry
RRC                   Rotate Right Circular
RRCA                  Rotate Accumulator Right Circular
RRD                   Rotate Accumulator and Memory Right Decimal
RST                   Restart
```

SET	Set Bit
SBC	Subtract with Carry (Borrow)
SCF	Set Carry Flag
SLA	Shift Left Arithmetic
SRA	Shift Right Arithmetic
SRL	Shift Right Logical
SUB	Subtract
XOR	Logical Exclusive OR

F.2 8080 OPCODES

Opcode	Function
ADC,ACI	Add with Carry
ADD,ADI	Add
ANA,ANI	Logical AND
CALL	Call Subroutine
CC	Call on Carry
CM	Call on Minus
CMA	Complement Accumulator
CMC	Complement Carry
CMP,CPI	Compare
CNC	Call on No Carry
CNZ	Call on Not Zero
CP	Call on Positive
CPE	Call on Parity Even
CPO	Call on Parity Odd
CZ	Call on Zero
DAA	Decimal Adjust
DAD	16-bit Add
DCR	Decrement
DCX	16-bit Decrement
DI	Disable Interrupts
EI	Enable Interrupts
HLT	Halt
IN	Input
INR	Increment
INX	Increment 16 bits
JC	Jump on Carry
JM	Jump on Minus
JMP	Jump
JNC	Jump on Not Carry
JNZ	Jump on Not Zero
JP	Jump on Positive
JPE	Jump on Parity Even
JPO	Jump on Parity Odd
JZ	Jump on Zero
LDA	Load Accumulator
LDAX	Load Accumulator Indirect
LHLD	Load HL Direct
LXI	Load 16 bits

MOV	Move
MVI	Move Immediate
NOP	No Operation
ORA,ORI	Logical OR
OUT	Output
PCHL	HL to Program Counter
POP	Pop from Stack
PUSH	Push to Stack
RAL	Rotate with Carry Left
RAR	Rotate with Carry Right
RC	Return on Carry
RET	Return from Subroutine
RLC	Rotate Left
RM	Return on Minus
RNC	Return on No Carry
RNZ	Return on Not Zero
RP	Return on Positive
RPE	Return on Parity Even
RPO	Return on Parity Odd
RRC	Rotate Right
RST	Restart
RZ	Return on Zero
SBB,SBI	Subtract with Borrow
SHLD	Store HL Direct
SPHL	HL to Stack Pointer
STA	Store Accumulator
STAX	Store Accumulator Indirect
STC	Set Carry
SUB,SUI	Subtract
XCHG	Exchange D and E, H and L
XRA,XRI	Logical Exclusive OR
XTHL	Exchange Top of Stack, HL

INDEX