8

8085/Z80 ASSEMBLER

USER'S MANUAL

XASM85 Assembler Version 1.04
Manual Revision 1.1 (22-Dec-83)

AVOCET SYSTEMS, INC.
804 SOUTH STATE ST.
DOVER, DELAWARE 19901

(302) 734-0151

CHAPTER 1. INTRODUCTION

This manual describes the features and operation of XASM85, Avocet Systems' assembler for the 8085 and Z80 microprocessors. In the introduction, you'll find an overview of the assembler's features, system requirements, and performance. There's also a list of skills and knowledge you should have before using the assembler, and a description of the syntax notation used in the rest of the manual.

1.1 Who This Manual Is Written For

This is a reference manual, written for experienced assembly-language programmers. It presents the information needed to use the assembler, in what we think is a clear and readable form. The manual is not intended to be a tutorial, but there are occasional tutorial asides for the benefit of newcomers. Also, we've tried to provide plenty of examples, to help you answer your own questions about how things work.

In writing this manual, we assumed that you:

-are reasonably familiar with your computer and its operating system (CP/M, MS-DOS, or equivalent).

-have a text editor, and know how to use it.

-know the architecture and instruction set of the 8085 and/or the Z80.

-have programmed in assembly language before.

1.2 Hints For Novices

If you lack any of the prerequisites listed above, you should plan to do some supplemental reading before you begin using the assembler (or at the same time, if you're really ambitious). The rest of this section discusses the prerequisites in more detail, and the next section suggests some books to read.

OPERATING SYSTEM: You can run the assembler without knowing your operating system intimately, but you may need to use some of its utility programs when the time comes to test or download an assembled program. For instance, CP/M's PIP and DDT programs are particularly useful; COPY and DEBUG are the equivalent commands under MS-DOS.

TEXT EDITOR: There's no way you can use the assembler without one; you have to have some way to get assembly-language programs into a disk file. If you're really new at this, and haven't used an editor before, you'll just have to bite the bullet and learn.

While we're on the subject, your editor should be one of the full-screen variety, such as VEDIT or WORDSTAR. Teletype-oriented editors such as CP/M's ED are guaranteed to be frustrating, although they can be used if nothing else is available.

THE 8085 CHIP: You will have to know the 8085's architecture and instruction set before you can write programs for it. However, you need not commit the entire instruction set to memory at once; learn a few instructions at a time, and begin using them. The assembler itself can help you, as it will reject most illegal instruction/operand combinations. This in turn should send you scurrying back to the 8085 Programming Manual (or equivalent) for review; eventually, you'll remember what's legal.

ASSEMBLY PROGRAMMING EXPERIENCE: If you haven't used an assembler before, don't panic; there's always a first time. Basic notions like "symbol", "expression", and "pseudo-instruction" may be unfamiliar to you, but you can probably make sense of them by reading this manual carefully. To fill in the gaps and see how these elements fit together, read or skim one of the books listed below.

1.3 Recommended Reading

A visit to your local computer store or university bookstore should turn up any number of books on the 8085 and Z80, and on the CP/M and MS-DOS operating systems. As a start, you might look at the ones listed here:

Fernandez, Judy N., and Ruth Ashley: 8080/8085 Assembly Language Programming. John Wiley & Sons, 1981

Spracklen, Kathe: Z-80 and 8080 Assembly Language Programming. Hayden Book Co., Inc.

Larsen, Titus, & Titus: 8080/8085 Software Design. Howard W. Sams Books.

In addition, we recommend the data books and programming manuals available from manufacturers of the 8085 and Z80 chips (eg. Intel, Zilog and Mostek). Even if you are experienced, it may be useful to have one of these on hand as a guide to the instruction set.

1.4 System Requirements

XASM85 is presently available under three different operating systems: CP/M-80, CP/M-86, and MS-DOS (PCDOS). There is a different version for each operating system. The CP/M-80 version works with CP/M versions 1.3 and higher, and also with extant versions of CDOS and TurboDos. It has been used successfully with several other CP/M look-alikes, though we cannot guarantee compatibility with unknown systems.

Under CP/M-80, XASM85 runs on 8080-, 8085-, and Z80-based computers. Approximately 17K bytes of memory are required for the assembler and internal buffers. Additional memory is used to store the assembly-time symbol table. Although the assembler can probably be run in a 32K system, we recommend 64K of memory.

Under CP/M-86, MS-DOS, or PCDOS, XASM85 runs on 8086- and 8088-based computers. Approximately 20K bytes of memory are required, with additional memory for the symbol table. The computer should have at least 64K of memory.

For all versions, at least one disk drive is required. We strongly recommend that you have a second disk drive and a printer.

1.5 Overview Of Assembler Operation

XASM85 accepts assembly language text from an input file on disk (often referred to as the <u>source</u> file). It generates two output files: The <u>object</u> file contains the machine language program produced by the assembler. The <u>listing</u> file contains an <u>assembly listing</u> - a copy of the input text, annotated to show the instruction codes and data generated by the assembler, and to indicate any errors detected. It also contains a table showing all symbols defined in the input text, and the values assigned to them.

The source language uses standard Intel mnemonics for the 8085 instruction set. In addition, it includes Intel-style ("TDL") mnemonics for the extended instruction set of the Z80.

The assembler normally paginates the assembly listing and symbol table. That is, it breaks them up into numbered pages for output to a printer. Each page begins with a page number, title, and subtitle.

The object file is in Intel "HEX" format. This format represents binary instruction codes and data as hexadecimal numbers in printable form. It minimizes file size by eliminating the need to store large blocks of initialized addresses, and it provides internal checksums to guard against file transmission errors.

During operation, the assembler displays certain information on the computer console. A sign-on banner, containing the assembler name, version, serial number, and copyright message, is displayed at the beginning of each run. Any source lines which contain errors are displayed as they are encountered, in assembly listing format. At the end of the run, the assembler tells you how many source lines contained errors, what fraction of the memory available at assembly-time was actually used, and the highest object address for which code or data was generated.

1.6 How Fast Is It?

XASM85 assembles approximately 4800 lines per minute while generating both an object file and an assembly listing. Without the assembly listing, this increases to 5900 lines per minute.

These figures were obtained by using the CP/M-80 version of the assembler, running under CP/M Version 2.2 on a 4 MHz. Z80 with a hard disk. The input file for the test was 2371 lines long, and defined about 100 symbols. The speed computation included the time required for the operating system to find and load the assembler, but not the time to re-boot the operating system upon completion.

Please note that assembler speed will vary substantially with the speed of the host computer. In particular, the assembler spends much of its time reading and writing disk files, and so is sensitive to disk access times and data rates. In general, you should expect it to be significantly slower with floppy disks than with the hard disk used in our tests.

1.7 Syntax Notation In This Manual

Throughout this manual, the following notation is used to describe the syntax of the assembly language and of assembler commands:

< >	Angle brackets enclosing lower-case text indicate that you must enter an item of a type defined by the text. For example, **<expr>** means you must enter an arithmetic expression.
<CR>	Angle brackets enclosing upper-case text indicate that you must press the key specified by the text. For example, **<CR>** means the **RETURN** key.
[ ]	Brackets indicate that the enclosed item is optional.
CAPS	Capital letters indicate items that must be entered exactly as shown except for case. That is, you may enter the same letters in lowercase.
...	An ellipsis indicates the preceeding item or sequence of items may be repeated zero or more times.
space	Where a space is shown, at least one space or tab character is required. Spaces and tabs may be used in combination, as many as you wish.
other	Other characters should be entered exactly as shown.

For example,

DB <expr>[,<expr>...]<CR>

means that you must enter the letters **DB** or **db**, followed by at least one space or tab, followed by one or more arithmetic expressions and the **RETURN** key. If you enter two or more expressions, they must be separated by commas.

Since it is understood that every line in a source file ends with a RETURN, we will generally omit the <CR> when describing the source language.

When we are presenting an example or specimen of an assembly-language construct, we will generally show it in boldface, as can be seen in the preceeding paragraphs.

CHAPTER 2. HOW TO OPERATE THE ASSEMBLER

This chapter describes the mechanics of using XASM85: how to invoke it, how to specify the file to be assembled, and how to control the generation and format of output files.

2.1 Getting Started

You can get started using the assembler even before you learn the details of its command-line syntax. You need only type its name, **XASM85**, followed by the name of the source file which you want processed. For example, to assemble the file **GRINCH.ASM**, you would type

```
XASM85 GRINCH<CR>
```

This instructs the operating system to start the assembler running, and to tell it to use GRINCH.ASM as the source file (the .ASM part is supplied by the assembler if you don't tell it otherwise).

2.2 XASM85 Command Lines

A complete XASM85 command line looks like this:

```
XASM85 [<drv>:]<name>[.<ext>] [<drv>:] [<flag>...]<CR>
```

Where:

<drv>	is a single character specifiying a disk drive (eg. A, B, etc.). The first <drv> specifies the drive on which the input file is located. The second <drv> specifies the drive on which the output files are to be placed. Either of the <drv>: specifiers may be omitted, in which case the current drive is used.
<name>	is a string of up to 8 characters, which is the name of the input file
<ext>	represents a file extension of up to three characters. It may be omitted, in which case the file extension "ASM" is used.

<flag>	is one of the single-character commands shown below, which control the operation of the assembler.

2.3 Command-Line Flags

The flags , if supplied, control various options as follows:

L	Listing Only (turn off object file)
O	Object Only (turn off assembly listing completely)
X	Turn off assembly listing (symbol table not suppressed)
Y	Turn off symbol table (assembly listing not suppressed)
C	Send listings to the console.
P	Send listings to the printer.
N	Suppress pagination of assembly listing; ie. no page headings or page ejects.
Q	Quiet. Turns off assembler sign-on banner.

Flags may be used in combination. For example, specifying "LCN" suppresses the object file, directs the listing to the console, and suppresses pagination. More examples are shown below.

2.4 Assembler Command Line Examples

Example 1. Command Without Flags

```
XASM85 GRINCH<CR>
```

The file GRINCH.ASM, located on the current drive, is assembled. Two output files are generated: GRINCH.HEX contains the object code, and GRINCH.PRN contains the assembly listing and symbol table listing.

Example 2. Complete Source File Specification

```
XASM85 C:GRINCH.FOO<CR>
```

The file GRINCH.FOO, located on the drive C, is assembled. The output files are the same as in Example 1, above. If we had left of the "C:", then GRINCH.FOO would have been sought on the current drive.

Example 3. Sending output files to a different drive.

```
XASM85 A:GRINCH B:<CR>
```

The file GRINCH.ASM, located on drive A, is assembled. The output files are the same as in the previous examples, but are placed on drive B.

Example 4. Sending the listing file to the printer.

```
XASM85 GRINCH PL<CR>
```

The file GRINCH.ASM is assembled, as before. This time, the listing file (containing the assembly listing and symbol table listing) is sent to your computer's LST: device, which presumably is the line printer. In addition, because we have included the "L" flag, no object file is generated.

Example 5. Getting A List Of Error Lines Only

```
XASM85 GRINCH LXY<CR>
```

GRINCH.ASM is assembled, as before. This time, we have turned off the object file with "L", suppressed the assembly listing with "X", and suppressed the symbol table listing with "Y". You might expect that the assembler would produce no output at all. However, the assembler always lists lines which contain errors, even if the listing is suppressed. Thus, the command line shown here will produce a .PRN file containing only those lines in which errors occurred. Since pagination is irrelevant here, you will probably want to suppress it also, thus:

```
XASM85 GRINCH LXYN<CR>
```

The order in which the flag characters appear is unimportant- the command

```
XASM85 GRINCH YXLN<CR>
```

would have the same effect. Finally, since "O" is equivalent to writing "X" and "Y", we could have written the command yet another way as

```
XASM85 GRINCH OLN<CR>
```

2.5 Summary of Defaults

If you don't specify otherwise in the command line, the assembler will behave as follows:

The source (input) file is sought on the current drive, and has extension ".ASM".

An object file is produced. It is placed on the current drive.

An assembly listing and a symbol table are generated. They are placed in a file on the current drive, with the same filename as the input file, and with extension ".PRN".

2.6 Aborting An Assembly

You can terminate an assembly at any time by striking a control-C at the console. This causes an exit to the operating system.

CHAPTER 3. SYNTAX OF STATEMENTS AND OPERANDS

An assembly-language program consists of a sequence of statements, each occupying one line of the source file. A statement comprises four fields, not all of which need be present: label, operation, operand, and comment. All statements are divided into these same fields in the same way, and thus share a common syntax. The allowed contents of the operand field vary according to the operation specified, but all operands are formed from a common set of elements: symbols, numbers, strings, arithmetic operators, etc.

Operations, and therefore statements, are of two kinds: instructions and pseudo-instructions. This section describes the syntax common to both kinds of statements, and the syntax of the elements used to form operands. The two subsequent sections describe instructions and pseudo-instructions in detail.

3.1 Statements

Each line of the source file contains a single assembly-language statement. A statement consists of four fields: label field, operation field, operand field, and comment field, arranged thus:

⌊<label>⌋ ⌊<operation> ⌊<operand>...⌋⌋ ⌊;<comment>⌋

The operation field contains a mnemonic symbol for an 8085 instruction or an pseudo-instruction (assembler directive). The exact interpretation of the label and operand fields varies according to which instruction or directive is present.

The label field, if present, consists of an identifier optionally followed by a colon. Using an identifier in the label field defines it as a user symbol and gives it a value. Ordinarily, this value is the address of the first byte of the instruction or data generated by the statement; that is, the symbol labels the current location. However, there are a few pseudo-instructions which assign other values to the symbol, independent of the current address. EQU is an example of such a pseudo-instruction.

The operand field consists of zero or more operands; if there are two or more, then they are separated by commas. There are several kinds of operands- registers, arithmetic expressions, and character strings, to name a few. Just which ones are allowed, and how many, depends on the operation specified.

The comment field consists of any sequence of printing characters, preceeded by a semicolon. It is intended for human consumption, and is always ignored by the assembler.

The fields are separated by whitespace; ie. by any combination of blanks and tabs. We suggest using tabs only. This not only saves space in the source file, but it allows you to line up the fields in vertical columns, making your program easier to read.

All of the fields are optional, except that operands may not be present unless there is an operation. Fields may begin in any column, except that-

-The label, if present, must begin in the first column.

-At least one blank or tab must precede the operation.

Blank lines are specifically allowed; they are treated as comments.

3.2 Examples Of Statements

Example 1. Line consisting only of a comment:

```
;this is a comment
```

Example 2. Instruction with label, operands, and comment:

```
FOO: LD   A,B        ;Load A from B
```

Example 3. Line consisting only of a label:

```
FOO:
```

Example 4. Line consisting only of an instruction:

```
LD   HL,1234H
```

Example 5. Pseudo-instruction defining a symbol:

```
MOO  EQU  5
```

3.3 Identifiers and Symbols

An identifier is a word or name, such as LDA, GRINCH, or A002. There are several kinds of identifiers in the assembly language: Mnemonics are the names of instructions and pseudo-instructions. Operators are the names of arithmetic functions, such as MOD (the remainder function) and XOR (bitwise exclusive-or). Symbols, as we use the term, are identifiers representing registers, addresses or numeric quantities, which may be used in forming operands.

Mnemonics, operators, and some symbols (for example the names of the 8085 registers), are pre-defined by the assembler. All other symbols must be defined by your program.

You can define an identifier as a symbol by using it as a label, or by using it in the label field of an EQU or DEFL statement. Identifiers may contain any of the following characters:

```
A..Z   a..z   0..9   $  .  _  ?
```

The first character of an identifier may not be a digit or dollar sign.

An identifier may consist of as many characters as you wish, but only the first 8 characters are significant. That is, two identifiers are the same if their first eight characters are the same. Also, the assembler makes no distinction between upper and lower case; for example, abcd and ABCD are the same identifier.

All of the pre-defined indentifiers are reserved; that is, you may not redefine them as symbols.

3.4 Numbers

A number consists of a sequence of digits, possibly including the hexadecimal digits A through F, optionally preceeded or followed by a character specifying the number base (radix). The first character must always be either a decimal digit (0-9) or a base specifier.

The base specifiers are:

base	leading specifier	trailing specifier
2	%	B
8	@	O or Q
16	$	H

If no base is specified, then base 10 is assumed.

The following examples all represent the number 127:

```
127
@177
177Q
$7F
7FH
%1111111
1111111B
```

Since a number must begin with a decimal digit or leading base specifier, you must be careful in writing hex constants that begin with a letter. For instance, the number 255 may be written as $FF or 0FFH, but not as FFH. The latter would be treated as a symbol by the assembler, resulting in a "U" (Undefined Symbol) error.

3.5 Character Constants

A character constant consists of one or two characters enclosed in single or double quotes (' or "). The single quote may be used as a character between double quotes, and vice-versa.

Character constants are evaluated as 16-bit integers, with each character converted to its ASCII code. For a single-character constant, the high-order byte is zero, and the low-order byte is the character code. For a two-character constant, the high-order byte of the value contains the first character code, and the low-order byte contains the second.

Thus, the following are equivalent:

```
'A'  "A"  and  41H

'AB' "AB" and  4142H
```

3.6 Location-Counter Reference

The assembly-time location counter, which keeps track of addresses in the generated code, may be referenced by the special symbol $ (dollar sign). The value of this symbol is the address of the first byte of code or data generated by the current statement. Thus, for example, the statements

```
ORG  4455H
DW   $,$
DW   $,$
```

would generate the following sequence of (hex) bytes, (remember that the low-order byte comes first):

```
55 44 55 44 59 44 59 44
```

3.7 Arithmetic Expressions

Symbols, numbers, character constants, and location counter references all evaluate to 16-bit integer values. Wherever such a value is allowed, you may also use an arithmetic expression, composed of one or more of these elements connected by operators (functions).

As you might expect, the set of functions includes + - * and / (addition, subtraction, multiplication, division). It also includes many other functions, such as modulo (remainder), left and right shift, and relational operators. All the functions except + - * and / are represented by identifiers, and must be separated from their arguments by at least one blank or tab.

Most of the operators treat their operands as 16-bit unsigned quantities. However, twos-complement negation, represented by a unary minus sign, is allowed. Negative quantities from -32768 to -1 can thus be written. They will give the expected results when added or subtracted from other quantities. However, no overflow checking is performed, and the results of other operations (including multiplication, division, and comparisons) do not take arithmetic sign into account. For example, -1 is equal to OFFFFH, the two's complement of 0001H. Thus,

```
-1 + 5 = 4
```

as expected, but

```
-1 LT 0
```

is an unsigned comparison of FFFF with 0, and is thus FALSE.

3.8 Operators Allowed In Expressions

Arithmetic Operators

+	Sum
-	Difference
unary +	+x is defined as 0+x
unary -	-x is defined as 0-x
*	Product (unsigned)
/	Quotient (unsigned)
MOD	Remainder (unsigned). x MOD y gives the remainder of x/y

Shift Operators

SHL	Binary left shift. x SHL y yields x shifted left y places (ie. x multiplied by 2^y).
SHR	Binary right shift, logical. x SHR y yields x shifted right y places (ie. x divided by 2^y). High-order bits are zero-filled.

If the right argument is negative, then the direction of the shift is reversed.

Byte-Extraction Operators

HIGH	Returns the value of the most significant byte of its argument.
LOW	Returns the value of the least significant byte of its argument

HIGH and LOW are unary operators, taking an argument on the right. For example: HIGH 1122H is 11H, and LOW 1122H is 22H.

Boolean Operators

NOT	Unary logical negation. Complements all the bits in its argument.
AND	Logical product; ie. each bit of the result is obtained by ANDing together the corresponding bits in the arguments.
OR	Logical sum.
XOR	Exclusive-OR.

These are all bitwise operators; that is, the same operation is performed on each operand bit position.

```
For example:   NOT 0 is OFFFFH
               101B AND 010B is 0
               101B OR  010B is 111B
               101B XOR 010B is 111B
               101B XOR 100B is 001B
```

Relational Operators

These perform unsigned 16-bit comparisons of their operands, returning 1 for TRUE and 0 for FALSE.

For comparison x R y, where R is a relational operator, the results are as follows (iff means "if and only if"):

EQ	TRUE iff x and y are equal
NE	TRUE iff x and y are not equal
LE	TRUE iff x is less than or equal to y
LT	TRUE iff x is strictly less than y
GE	TRUE iff x is greater than or equal to y
GT	TRUE iff x is strictly greater than y.

3.9 Evaluation of Expressions

The order in which parts of an arithmetic expression are evaluated is governed by precedence values assigned to the operators. Precedence may be thought of as "tightness of binding"; operators with higher precedence bind more tightly to their operands, and thus are evaluated first. For example,

```
3 * 5 + 4
```

is interpreted as

```
(3*5) + 4
```

because the multiplication operator has higher precedence than the addition operator. In general, precedence values have been chosen to coincide with your intuitive notions of how expressions should be read. A table of relative operator precedences is given on the next page.

When two operators have the same precedence, the expression is evaluated from left to right. Thus,

```
4-1+2
```

is interpreted as

```
(4-1)+2
```

You can override the assumed order of evaluation by using parentheses in the normal way. In fact, we recommend that you do so in all but the simplest cases. Not only will this leave no doubts about what you meant, but it will prevent portability problems should you later use a different assembler. (other assemblers may assign different relative precedence to the more obscure operators).

TABLE 1. RELATIVE PRECEDENCE OF OPERATORS

Groups of operators are shown in order of descending precedence. All operators in a group (ie. on the same line) have the same precedence.

unary +, unary -	(HIGHEST PRECEDENCE)
HIGH LOW	
* / MOD SHR SHL	
+ -	
EQ NE LT LE GT GE	
NOT	
AND	
OR XOR	(LOWEST PRECEDENCE)

3.10 Forward References

A forward reference is a reference to a symbol which is not defined until later in the source program. For example, take the following program fragment:

```
              DS   GRINCH+1
     GRINCH   EQU  7
              DS   GRINCH+1
```

The use of GRINCH in the first DS statement is a forward reference, since GRINCH has not yet been defined when this statement is encountered. The use of GRINCH in the second DS is not a forward reference, as the statement defining GRINCH has already been encountered.

There are a number of places in the assembly language where forward references, and expressions containing them, are specifically prohibited. The general rule is that a forward reference is not allowed where the value of the expression affects the location counter or controls the number of bytes of code generated. Most cases where this occurs are in pseudo-instructions; if forward references are not allowed, this will be stated in the pseudo-op description. The assembler will report an "F" error whenever it finds an illegal forward reference.

To further clarify the notion of a forward reference, consider the following statement:

```
          JMP  $+5
```

The expression $+5 is not a forward reference, even though the jump is forward, because the value of the location counter ($) is already known when the JMP statement is processed.

CHAPTER 4. THE 8085/Z80 INSTRUCTION SET

This chapter describes the assembly-language syntax of 8085 and Z80 instructions. A complete discussion of 8085/Z80 architecture and instruction semantics (ie. what the instructions do) is beyond the scope of this manual; for this information, see any of the books mentioned under Recommended Reading in the Introduction.

4.1 CPU Registers

The assembly language includes pre-defined symbols for all of the CPU registers that may be referenced in instructions. These symbols do not have numeric values, and may be used only where a register is specifically allowed. However, their names are reserved; they may not be re-defined as user symbols.

The 8-bit registers are:

```
     A  B  C  D  E  H  L
```

The names

```
     B  D  H
```

are also used to refer to the 16-bit register pairs consisting of BC, DE, and HL. Other 16-bit registers are named as follows:

```
     SP  PSW  X  Y
```

Also, the memory location addressed by the contents of the HL register pair may be referred to by the name M, which behaves syntactically like an 8-bit register name (in most contexts).

4.2 Immediate and Direct Addressing

Direct addressing uses an instruction operand as the address of a data item to be operated upon. Immediate addressing uses the operand itself as data. In both cases the operand is syntactically just a numeric value (ie. an arithmetic expression); the addressing mode depends upon the instruction mnemonic. For instance,

```
          LXI  H,1000
```

loads HL with the immediate value 1000;

```
          LHLD 1000
```

loads HL with the 16-bit contents of memory locations 1000,1001.

4.3 Indexed Addressing

An operand address may be computed as the sum of a signed displacement and the contents of an index register (IX or IY). This is denoted by one of the forms

```
     <d>(X)    <d>(Y)
```

where <d> is an expression in the range -128..+127. For example, if index register X contains the value 1000H, then

```
     MOV  A,2(X)
```

loads the A register with the contents of location 1002H.

4.4 8-Bit Load Instructions

MOV	<reg>,<reg>	Load register from register.
MVI	<reg>,<imm8>	Load register with immediate value.
MOV	<reg>,M	Load register from memory, indirect
MOV	<reg>,<ndx>	*Load register from memory, indexed.
MOV	M,<reg>	Store register to memory, indirect.
MOV	<ndx>,<reg>	*Store register to memory, indexed.
MVI	M,<imm8>	Load memory immediate, indirect.
MVI	<ndx>,<imm8>	*Load memory immediate, indexed.
LDAX	B	Load A indirect via BC.
LDAX	D	Load A indirect via DE.
LDA	<addr>	Load A from memory, direct.
STAX	B	Store A indirect via BC.
STAX	D	Store A indirect via DE.
STA	<addr>	Store A to memory, direct.
LDAI		*Load A from interrupt register.
LDAR		*Load A from refresh register.
STAI		*Store A to interrupt register.
STAR		*Store A to refresh register.

SYNTAX NOTATION

In the instruction-set description:

<reg> is any of the 8-bit registers: A B C D E H L

<rp> is any of the register pairs: B D H SP

<addr> is any 16-bit quantity.

<imm8> is any quantity in the range -128..255

<ndx> is d(X), or d(Y)
where d is any quantity in the range -128..+127.

* indicates Z80-only instruction.
# indicates 8085-only instruction.

4.5 16-Bit Load Instructions

LXI <rp>,addr	Load immediate word.
LXI X,addr	*
LXI Y,addr	*
LHLD <addr>	Load word from memory, direct.
LBCD <addr>	*
LDED <addr>	*
LSPD <addr>	*
LIXD <addr>	*
LIYD <addr>	*
SHLD <addr>	Store word to memory, direct.
SBCD <addr>	*
SDED <addr>	*
SSPD <addr>	*
SIXD <addr>	*
SIYD <addr>	*
SPHL	Load stack pointer from register
SPIX	*
SPIY	*

4.6 Push and Pop Instructions

PUSH PSW	Push 16-bit register on stack.
PUSH B	
PUSH D	
PUSH H	
PUSH X	*
PUSH Y	*
POP PSW	Pop stack top into 16-bit register.
POP B	
POP D	
POP H	
POP X	*
POP Y	*

4.7 Exchange Instructions

XCHG	Exchange DE and HL.
EXAF	*Exchange AF with alternate AF.
EXX	*Exchange with alternate registers.
XTHL	Exchange stack top with HL.
XTIX	*Exchange stack top with IX.
XTIY	*Exchange stack top with IY.

4.8 Block Move and Block Compare Instructions

LDI	*Move byte and increment pointers.
LDIR	*Move byte, increment, and repeat.
LDD	*Move byte and decrement pointers.
LDDR	*Move byte, decrement, and repeat.
CCI	*Compare byte and increment pointers.
CCIR	*Compare byte, increment, and repeat.
CCD	*Compare byte and decrement pointers.
CCDR	*Compare byte, decrement, and repeat.

4.9 8-Bit Arithmetic Instructions

ADD <reg>	Add to accumulator.
ADI <imm8>	
ADD M	
ADD <ndx>	*
ADC <reg>	Add to accumulator,
ACI <imm8>	
ADC M	
ADC <ndx>	*
SUB <reg>	Subtract from accumulator
SUI <imm8>	
SUB M	
SUB <ndx>	*
SBB <reg>	Subtract from accumulator,
SBI <imm8>	with borrow.
SBB M	
SBB <ndx>	*
ANA <reg>	Logical AND with accumulator
ANI <imm8>	
ANA M	
ANA <ndx>	*
ORA <reg>	Logical ORA with accumulator
ORI <imm8>	
ORA M	
ORA <ndx>	*
XRA <reg>	Exclusive OR with accumulator
XRI <imm8>	
XRA M	
XRA <ndx>	*

CMP	<reg>	Compare with accumulator
CPI	<imm8>	
CMP	M	
CMP	<ndx>	*
INR	<reg>	Increment register.
INR	M	Increment memory, indirect.
INR	<ndx>	*Increment memory, indexed.
DCR	<reg>	Decrement register.
DCR	M	Decrement memory, indirect.
DCR	<ndx>	*Decrement memory, indexed.

4.10 Control Instructions

DAA	Decimal Adjust Accumulator.
CMA	Complement accumulator.
NEG	*Negate Accumulator.
CMC	Complement Carry Flag.
STC	Set Carry Flag.
NOP	No-Op.
HLT	Halt processor.
DI	Disable Interrupts.
EI	Enable Interrupts.
IM0	*Set interrupt mode.
IM1	*
IM2	*
RIM	#
SIM	#

4.11 16-Bit Arithmetic Instructions

DAD	<rp>	Add to HL.
DADC	<rp>	*Add to HL with carry.
DSBC	<rp>	*Subtract from HL with borrow.
DADX	BC	*Add to X.
DADX	DE	*
DADX	IX	*
DADX	SP	*
DADY	BC	*Add to IY.
DADY	DE	*
DADY	IY	*
DADY	SP	*
INX	<rp>	Increment 16-bit register.
INX	X	*
INX	Y	*

DCX <rp>	Decrement 16-bit register.
DCX X	*
DCX Y	*

4.12 Shift and Rotate Instructions

RLC	Rotate A left.
RAL	Rotate A left thru carry.
RRC	Rotate A right.
RAR	Rotate A right thru carry.
RLCR <reg>	*Rotate register left.
RLCR M	*Rotate memory left, indirect.
RLCR <ndx>	*Rotate memory left, indexed.
RALR <reg>	*Rotate register left thru carry.
RALR M	*Rotate memory left thru carry, indirect.
RALR <ndx>	*Rotate memory left thru carry, indexed.
RRCR <reg> g	*Rotate register right.
RRCR M	*Rotate memory right, indirect.
RRCR <ndx>	*Rotate memory right, indexed.
RARR <reg>	*Rotate register right thru carry.
RARR M	*Rotate memory right thru carry, indirect.
RARR <ndx>	*Rotate memory right thru carry, indexed.
SLAR <reg>	*Shift Left Arithmetic, register.
SLAR M	*Shift Left Arithmetic, memory, indirect.
SLAR <ndx>	*Shift Left Arithmetic, memory, indexed.
SRAR <reg>	*Shift Right Arithmetic, register.
SRAR M	*Shift Right Arithmetic, memory, indirect.
SRAR <ndx>	*Shift Right Arithmetic, memory, indexed.
SRLR <reg>	*Shift Right Logical, register.
SRLR M	*Shift Right Logical, memory, indirect.
SRLR <ndx>	*Shift Right Logical, memory, indexed.
RLD	*Rotate Left, Digit.
RRD	*Rotate Right, Digit.

4.13 Bit-Manipulation Instructions

```
     BIT  <bitno>,<reg>       *Test bit in register.
     BIT  <bitno>,M           *Test bit in memory, indirect.
     BIT  <bitno>,<ndx>       *Test bit in memory, indexed.

     SET  <bitno>,<reg>       *Set bit in register.
     SET  <bitno>,M           *Set bit in memory, indirect.
     SET  <bitno>,<ndx>       *Set bit in memory, indexed.

     RES  <bitno>,<reg>       *Reset bit in register.
     RES  <bitno>,M           *Reset bit in memory, indirect.
     RES  <bitno>,<ndx>       *Reset bit in memory, indexed.
```

<bitno> is any quantity in the range 0..7

4.14 Jump, Call, and Return Instructions

JMP	<addr>	Jump unconditionally.
JZ	<addr>	Jump if zero
JNZ	<addr>	Jump if not zero
JC	<addr>	Jump if carry
JNC	<addr>	Jump if no carry
JPE	<addr>	Jump if parity even
JPO	<addr>	Jump if parity odd
JO	<addr>	*Jump if overflow
JNO	<addr>	*Jump if no overflow
JP	<addr>	Jump if plus
JM	<addr>	Jump if minus
JMPR	<addr>	*Jump unconditionally, relative.
JRC	<addr>	*Jump if carry, relative.
JRNC	<addr>	*Jump if no carry, relative.
JRZ	<addr>	*Jump if zero, relative.
JRNZ	<addr>	*Jump if not zero, relative.
PCHL		Jump indirect via HL
PCIX		*Jump indirect via X
PCIY		*Jump indirect via Y
DJNZ	<addr>	*Decrement and jump if not zero.
CALL	<addr>	Call unconditionally.
CZ	<addr>	Call if zero
CNZ	<addr>	Call if not zero
CC	<addr>	Call if carry
CNC	<addr>	Call if no carry
CPE	<addr>	Call if parity even
CPO	<addr>	Call if parity odd
CO	<addr>	*Call if overflow
CNO	<addr>	*Call if no overflow
CP	<addr>	Call if plus
CM	<addr>	Call if minus
RST	0	Restart (special call).
RST	1	
RST	2	
RST	3	
RST	4	
RST	5	
RST	6	
RST	7	

RET	<addr>	Return unconditionally.
RZ	<addr>	Return if zero
RNZ	<addr>	Return if not zero
RC	<addr>	Return if carry
RNC	<addr>	Return if no carry
RPE	<addr>	Return if parity even
RPO	<addr>	Return if parity odd
RO	<addr>	*Return if overflow
RNO	<addr>	*Return if no overflow
RP	<addr>	Return if plus
RM	<addr>	Return if minus
RETI		*Return from interrupt.
RETN		*Return from non-maskable interrupt.

4.15 Input and Output Instructions

IN	<port>	Input to accumulator.
INP	<reg>	*Input indirect via C.
INI		*Input and increment.
INIR		*Input, increment, and repeat.
IND		*Input and decrement.
INDR		*Input, decrement, and repeat.
OUT	<port>	Output from accumulator.
OUTP	<reg>	*Output indirect via C.
OUTI		*Output and increment.
OUTIR		*Output, increment, and repeat.
OUTD		*Output and decrement.
OUTDR		*Output, decrement, and repeat.

<port> is any quantity in the range 0..255

CHAPTER 5. PSEUDO-INSTRUCTIONS

Pseudo-instructions are commands to the assembler, which look syntactically like machine instructions. For example, to define a symbol S55 with the value 1000H, you could write

```
        S55  EQU  1000H
```

EQU is a pseudo-instruction which EQUates the symbol to the value of the expression on the right.

Pseudo-instructions are also called "pseudo-operations", "pseudo-ops", or "directives". We will use all of these terms interchangeably.

The remainder of this chapter describes the various pseudo-instructions in detail.

5.1 Storage Definition

Define Bytes

DB <arg>[,<arg>...]

DB reserves a series of single-byte locations, initialized according to the values of its operands. Each <arg> may be either an expression or a string. Expressions must evaluate to 8-bit values (high byte either 0 or 255). Strings look just like character constants, except that a string may contain more than two characters. A string may be enclosed in single or double quotes.

For each expression, a single byte of storage is reserved, initialized to the low byte of the expression's value. A Range Error ("R") occurs if any of the expressions is not in the range -128..255.

For each string, the characters of the string are stored in sequential reserved bytes.

If a compound expression beginning with a character constant is used in a DB, then the expression must be enclosed in parentheses to keep it from being incorrectly parsed as a string. For example,

```
        DB ('A'+1)
```

legitimately sums a character constant and a number, but

```
        DB 'A'+1
```

causes a syntax error (the operand is parsed as a 1-character string followed by some garbage).

Define Words

DW <expr>[,<expr>...]

DW reserves a series of two-byte locations, initialized to the values of its operands. For the 8085, the low byte of the each value is placed first, and the high byte second.

For example, the statement

```
DW   2233H, 4455H
```

reserves and initializes four bytes, as follows:

```
33H  22H  55H  44H
```

Define Space

DS <expr>

DS reserves a block of n bytes, where n is the value of the expression. The bytes are not initialized. The expression may not contain any forward references.

For example, if the value of COUNT is 4, then

```
DS     COUNT+1
```

reserves five bytes.

5.2 Program Origin

ORG <expr>

The ORG statement sets the program origin- that is, the starting value of the location counter- to the value of its operand expression. This sets the beginning address of the code and data which follows.

Every program should have an ORG before the first code-generating statement. Additional ORG statements may be used to produce program segments which will load at different locations.

The expression in the operand field may not contain any forward references.

If the source file contains no ORG statements, the location counter will initially be set to zero.

5.3 End of Program

END [<expr>]

The END statement informs the assembler that the end of your source program has been reached. It must be present, and must be the last statement in the program.

The operand expression, if any, specifies a program start address to be included in the object file. This is the address to which a loader should transfer control after it loads the object code into memory. If no start address is specified, then the value 100H is used.

If you forget to include an END statement in your source file, the assembler will supply one. A warning message will be displayed on the console.

If for some reason you have statements after the END statement, they will be completely ignored by the assembler.

5.4 Symbol Definition

<symbol> EQU <expr>

EQU defines a symbol with a specified value. For example,

```
        ZORCH     EQU       22
```

creates a symbol named ZORCH, whose value is 22. Symbols defined with EQU are treated as constants; any attempt to re-define such a symbol will cause an error. In general, you may write

```
        symbol     EQU        expression
```

where the symbol on the left has not been previously defined. The expression on the right may not contain any forward references.

<symbol> DEFL <expr>
<symbol> ASET <expr>

DEFL is used to define symbols, just like EQU. The only difference is that symbols defined with DEFL may have their values changed by subsequent DEFL's, without causing an error. ASET is a synonym for DEFL, and behaves identically.

For example, the sequence of statements:

```
        XYZ  DEFL      1
             DB        XYZ
        XYZ  DEFL      5
             DB        XYZ
```

would leave XYZ set to 5, and would generate the same data as

```
        DB    1,5
```

5.5 Offset Assembly

LOC
ENDLOC

The LOC and ENDLOC pseudo-instructions allow you to assemble code which will be moved to another address before execution. For example, you might have a ROM-resident routine which will be copied into RAM and executed there.

A sequence of statements beginning with LOC and ending with ENDLOC will produce code that loads in-line as if the LOC and ENDLOC were not present. However, the code will be assembled to execute at the address given in the LOC statement. More specifically, the location counter- and consequently the values of $ and of any labels defined- will reflect the LOC address.

An example may help to clarify this. Suppose we have the following program fragment:

```
              ORG   100H
FOO:          LD    A,B

              LOC   5000H
BAR:          DEC   B
              JNZ   BAR
MOO:          RET
              ENDLOC

ZOT:          DB    0
```

When this sequence of statements is assembled and loaded, the LD A,B instruction will be found at 100H, the DEC B at 101H, the JNZ at 102H, and so on. However, the values of the labels will be: FOO=100H, BAR=5000H, MOO=5004H, and ZOT=101H. Thus, the statements between LOC and ENDLOC will execute correctly if (and only if) their machine code is moved to location 5000H.

The effect of a LOC is cancelled when an ORG is encountered; both the load and execution location counters are set to the address specified in the ORG. Thus you must always set the load address first, using ORG, before setting the execution address with LOC.

The expression in the LOC statement may not contain forward references.

5.6 Conditional Assembly

IF, ELSE, and ENDIF
COND and ENDC

Conditional assembly allows the value of a symbol to control whether or not a particular group of statements is assembled.

The construct

```
IF <expr>
statement
.
.
.
ELSE
statement
.
.
.
ENDIF
```

behaves as follows: If the value of the expression is non-zero, then the statements between the IF and the ELSE are assembled. Otherwise, these statements are ignored, and the statements between ELSE and ENDIF are assembled. The ELSE and statements following it may be omitted.

The assembler recognizes COND as a synonym for IF, and ENDC as a synonym for ENDIF.

Conditionals may be nested to a depth of 10. The expression in the IF statement may not contain any forward references.

Each statement within a conditional block must be syntacticaly correct, at least to the extent of having the fields properly delimited and a valid instruction field. This is because the assembler must parse each statement in order to look for an ELSE or ENDIF, even if the block is not being assembled.

5.7 General Listing Control

New Page

```
PAGE [<expr>]
EJECT [<expr>]
```

If the <expr> is omitted, PAGE causes an immediate skip to the top of the next page. If <expr> is present and has value n, then a skip occurs only if less than n lines remain on the current page. EJECT is synonymous with PAGE. The PAGE or EJECT statement itself is not shown in the assembly listing.

Page Width

```
WIDTH <expr>
```

Sets the width of the listing page to the specified value, which may range from 32 to 132. The WIDTH statement itself is not shown in the listing.

Page Length

```
PGLEN <expr>
```

Sets the number of lines which will be printed on each page of the listing; this may range from 8 to 255. Note that seven of the lines are occupied by the page heading. The PGLEN statement itself is not shown in the listing.

LISTING ON/OFF

LIST and NOLIST

These allow selective listing of portions of a program. NOLIST turns off the assembly listing, and LIST turns it back on. If listing has been turned off with NOLIST, then the next LIST encountered will begin a new page. Command-line switches which disable listing (ie. X and O) will take precedence over LIST. NOLIST does not turn off listing of the symbol table.

LIST and NOLIST are not themselves shown in the assembly listing.

5.8 Title and Subtitle

TITLE <dlm><text><dlm>

Causes the specified text to become the listing page title, beginning with the next page header printed. The delimiter <dlm> may be any single printing character which does not appear in the enclosed text. For example, using single-quote as a delimiter, you could write:

```
        TITLE     '8085 Assembler Demo Program'
```

If the text contains a single quote, you could delimit it with slashes instead, as in:

```
        TITLE     /Avocet's Demo Program/
```

Note that TITLE is a pseudo-op, and so must appear in the operation field. If you put it in the label field (ie. begin it in column 1), it will not be recognized.

If no TITLE statement is used, then XASM85 supplies a default title-

```
        SOURCE FILE NAME: <name>.<ext>
```

giving the name and extension of the source file.

SBTTL <dlm><text><dlm>

SBTTL has the same syntax as TITLE, but it sets the page subtitle. The subtitle is printed on the line following the title line.

If no SBTTL statement is present, then the subtitle line will be left blank.

5.9 External Source Files

INCLUDE <filename>

INCLUDE inserts the contents of the specified file into the source text. The syntax of <filename> follows standard CP/M and MS-DOS conventions: A complete filename has the form

```
[<drv>:]<name>[.<ext>]
```

where <drv> is a single letter specifying a disk drive (A, B, etc); <name> is an alphanumeric file name of up to 8 characters; and <ext> is an alphanumeric file extension of up to 3 characters.

INCLUDE statements may not be nested; that is, the file inserted may not contain another INCLUDE. Ordinarily, the entire file is read; however, if an END statement is encountered, then INCLUDE processing terminates and text input reverts to the main source file.

5.10 Target Microprocessor Validation

.8080
.Z80

These pseudo-ops inform the assembler that the target microprocessor is other than an 8085. Any instructions not available in the specified processor will be flagged as "W" errors to alert you that they will not run correctly.

In the absence of either of these directives, all of the 8085 instructions are valid; instructions which are available only in the Z80 will be flagged.

When .8080 is specified, only the 8080 subset is valid. This subset consists of all the 8085 instructions except RIM and SIM.

When .Z80 is specified, the full Z80 instruction set is valid. This includes all the 8085 instructions except RIM and SIM, along with all those instructions peculiar to the Z80.

CHAPTER 6. ERROR HANDLING

Two types of errors can occur during an assembly: fatal and non-fatal. Fatal errors represent conditions which prevent the assembler from continuing. For instance, running out of disk space for an output file would cause a fatal error. Non-fatal errors represent faults detected in the source text, which do not require termination of the assembly.

When a fatal error occurs, assembly is aborted, and a descriptive error message is printed on the console. Control returns immediately to the operating system. Some typical causes of fatal errors are: missing source file; insufficient disk space; insufficient memory; or overflow of various stacks internal to the assembler.

When a non-fatal error occurs, the source line containing the error is flagged with a character in the first column of the assembly listing (this column is otherwise blank). Lines containing errors are always listed, even if listing is turned off. In addition, such lines are always displayed on the console as they are encountered.

There are some non-fatal errors which cannot be associated with a particular source line. These errors are always noted by a message on the console.

A message giving the total number of erroneous lines is printed at the end of the assembly listing, and displayed on the console. It will say either

```
          ***** nnnn LINES CONTAINED ERRORS *****

  or      ***** NO ERRORS WERE DETECTED *****
```

Only one error is listed per line; hence, if a line contains multiple errors some may not be caught until successive assembler runs.

Some non-fatal errors not associated with a particular source line are indicated by messages on the console.

The error messages and error flag characters are described in Appendix A.

CHAPTER 7. FORMAT OF MESSAGES AND LISTINGS

7.1 Assembler Sign-On Banner

When the assembler begins executing, it displays the following "banner" on the console:

```
-----------------------------------------------------------
  AVOCET 8085/Z80 ASSEMBLER - Version 1.02 - Serial #00100
  Copyright (C) 1983 by Avocet Systems, Inc.
  All Rights Reserved.
-----------------------------------------------------------
```

The version and serial number may differ from the ones shown here, as they represent your individual copy of the assembler. If you get tired of seeing the banner, you can suppress it by specifying the "Q" (Quiet) flag in the assembler command line.

7.2 Final Messages

At the end of each run, you will see two additional messages on the console:

```
USE FACTOR:           xxx%
HIGHEST ADDRESS USED: aaaaH
```

The Use Factor is the fraction of available host memory used at assembly time, expressed as a percentage. For example, a Use Factor of 15% means that 85% of the host computer's memory was unused. This represents only the memory occupied by user-defined symbols; if your program defines no symbols, then the Use Factor will be zero.

The Highest Address message refers to the target memory space. The address given is the highest one into which a data or code byte will be placed when then assembled program is loaded.

7.3 Page Headings

Paginated listings begin with a heading consisting of seven lines:

```
(blank line)
(blank line)
(assembler name and version)
(blank line)
(title and page number)
(subtitle)
(blank line)
```

If no title is supplied in the source program, then the assembler provides a default title consisting of the message

```
SOURCE FILE NAME: <name>.<ext>
```

where **<name>** is the file name, and **<ext>** is the file extension. The page number is listed at the right-hand end of this same line, always within the specified page width.

If no subtitle is supplied, the subtitle line is left blank. Both title and subtitle will be truncated, if necessary, to satisfy page width constraints.

7.4 Line Headings

Each code-generating line of the listing begins with the error flag (blank if no error) and the 4-digit hexadecimal value of the location counter as of the start of the line. This is followed by up to four bytes of generated code, also in hexadecimal with two digits per byte. Statements which generate more than 4 bytes will be assembled correctly, but only the first four bytes are listed.

Lines which do not generate code but which evaluate an operand (such as EQU) list the operand value in their header, in place of the location counter.

Lines containing listing-control pseudo-ops, such as WIDTH and TITLE do not appear at all in the listing.

7.5 Symbol-Table Listing

The symbol-table listing shows all symbols defined in the current assembly, with their hexadecimal values. Only user-defined symbols are listed. Symbols are in vertical columns, sorted alphabetically according to the ASCII collating sequence. The number of columns is adjusted automatically to fit in the specified page width. All pages of the symbol-table listing are automatically subtitled:

```
---- SYMBOL TABLE ----
```

Because the sorting scheme "alphabetizes" symbol names even if they end in numeric characters, the listing order may not be what you expect. For example, a typical sequence of symbols might appear as follows:

```
SYM19
SYM2
SYM20
SYM21
SYM3
SYM4
```

APPENDIX A. ERROR MESSAGES AND FLAGS

Non-Fatal Error Flags

These flag characters will appear in the first column of the assembly listing, to mark lines containing errors. The first column is otherwise blank.

C	Conditional Err	Unmatched IF, ELSE, or ENDIF; or conditionals nested too deep.
F	Fwd Ref Err	Illegal forward reference.
I	INCLUDE Err	File not found, or nested INCLUDEs.
M	Multiple Defn	Symbol already defined.
O	Operator Err	Undefined or illegal operator.
P	Phase Err	Symbol had different value on Pass 2 than on Pass 1.
R	Range Err	Operand out of range (address or value)
S	Syntax Err	Ill-formed argument or expression.
U	Undefined	Undefined symbol(s) in operand field.
W	Warning	Instruction or mode not valid for chip specified (eg. attempt to use a Z80-only instruction on the 8085).

Non-Fatal Error Messages

These messages are displayed on the console when the assembler encounters a non-fatal error condition which cannot be associated with any particular line in the source file.

NO ROOM FOR SYMBOL-TABLE SORT	Not enough memory is available to sort the symbol table; the symbol-table listing is omitted.
END STATEMENT INSERTED COURTESY OF AVOCET SYSTEMS	The assembler reached the end of the source file without seeing an END statement. This could mean that part of your program is missing. The assembler supplies an END statement, which is marked by a comment indicating where it came from.

Fatal Error Messages

When a fatal error occurs, assembler operation is immediately terminated, and control returns to the operating system. One of the following explanatory messages will be displayed on the console:

SOURCE FILE NOT FOUND	The specified source file doesn't exist.
UNABLE TO CREATE OUTPUT FILE	The directory is full on the disk specifed for output.
OUTPUT FILE WRITE ERROR	The output disk is full.
EVALUATION STACK FULL	An arithmetic expression was encountered which had too many levels of parentheses or of precedence nesting.
SYMBOL TABLE FULL	Not enough memory remains to create a table entry for a symbol being defined.

APPENDIX B. OBJECT FILE FORMAT

Object files are in the Intel HEX format, which represents binary data bytes as two-digit ASCII hexadecimal numbers. An object file consists of a sequence of data records, followed by a single end record.

The record formats are:

Data Record:

Byte 1	Colon (:)
2..3	Number of binary data bytes in this record.
4..5	Load address for this record, high byte.
6..7	Load address, '' '' '' low byte.
8..9	Record Type: "00"
10..x	Data bytes, two characters each.
x+1..x+2	Checksum (2 characters).
x+3..x+4	CR/LF

End Record:

Byte 1	Colon (:)
2..3	"00"
4..7	Program execution address, if specified in assembler END statement; else "0000".
8..9	Record Type: "01"
10..11	Checksum ("FF" if load address is all zeros)
x+3..x+4	CR/LF

The checksum is the two's complement of the 8-bit sum, without carry, of all the data bytes, the two bytes of load address, and the byte count.

APPENDIX C. SOURCE FILE PREPARATION

You can use almost any text editor or word processor to prepare source files for the assembler. We use both VEDIT and WORDSTAR, and have also used ED and EDLIN on occasion. However, you should be sure that your editor terminates lines properly, and that it does not put any "funny" characters into the file. If you're uncertain about this, read the next couple of paragraphs, and then go ahead and try it. If there are any problems, they should show up right away.

If you see assembly errors with no apparent cause, or find lines missing from the assembly listing, you should check the source file. Frequently, an offending line can be fixed by deleting and re-typing it (you may have to do the same for the lines immediately before and after it).

If you use WORDSTAR to edit source files, be careful always to use its "non-document" mode. Use of the "document" mode, may insert spurious characters (typically spaces and linefeeds with the high-order bit set) into the file; these characters will cause assembly problems. The same advice holds for other word processors: If there's a non-document mode, use it.

Each line in the source file should end with a RETURN and a LINEFEED, in that order. An EOF (hex 1A) must follow the end of the last line in the file. (Most editors take care of these special characters automatically). No other non-printing characters should appear in the file, except within comments. The SPACE and TAB characters may be used interchangeably or in combination, wherever whitespace is allowed.

APPENDIX D. ESTIMATING MEMORY REQUIREMENTS

In addition to space occupied by the assembler itself, assembly-time memory is used for file i/o buffers, and to store user-defined symbols. At most 4K (bytes) is required for buffers. Each symbol occupies n+4 bytes, where n is the number of characters in the symbol's name. An additional two bytes per symbol are required near the end of the assembly run if a listing of the symbol table is produced; this extra memory is used in sorting the symbol table.

Let's calculate, approximately, the number of symbols which could be defined in a typical CP/M-80 system with 64K (65536 bytes) of memory. We'll make it a worst-case calculation by assuming that every symbol is of maximum length (8 characters) and that a full 4K is required for file buffers:

Total Memory Available	65,000 bytes
Less Memory Used By CP/M	-15,000

User Memory Available	50,000 bytes
Less memory required for assembler	-12,500
Less memory required for buffers	-4,200

Memory Available For Symbol Table.......	33,300 bytes

$$\frac{33{,}300 \text{ bytes}}{8+4+2 \text{ bytes/symbol}} = \frac{33{,}300}{14} = 2{,}378 \text{ symbols}$$

The situation is actually even better than these figures indicate, for several reasons. First, not all symbols will be of maximum length. Second, file buffers are dynamically allocated, so not all of the 4K buffer space is needed throughout the assembly. And finally, an assembly can be run successfully even if there is insufficient space to sort the symbol table, at the expense of losing the symbol-table listing.