CHAPTER 5

USING THE ASSEMBLER

The **x80** cross assembler translates your assembly language source files into object images. The C cross compiler calls **x80** to assemble your code automatically, unless you specify otherwise.

The Programming Environment

x80 generates relocatable code. Documentation in the form of listings and symbol tables may also be produced.

Code is considered relocatable when the linker, **lnk80,** can process it for execution at various addresses in the target processor's memory. Relocatable code is used when a program is written as a number of separate modules, with each module presenting a standard interface and with local variables hidden from other modules. Each module is assembled separately, and the linker combines the object modules to produce the final, executable code.

Library Support

A library is a file containing modules of relocatable code for general use. A librarian is a utility for building and maintaining libraries. The librarian included with the C cross compiler is called **lby.** The **lord80** utility, for ordering the modules in a library, is also provided. See Chapter 8 for information on **lby** and **lord80.**

Linker Support

Modules of relocatable code sections may be combined by the linker to form an executable binary file. The linker will also search libraries and include modules from them, as required to resolve undefined symbols. The linker included with the C cross compiler is called **lnk80.** See Chapter 6 for information on using the linker.

Other Related Utilities

Other utilities included with the cross compiler are **hex80** and **rel80.** **hex80** converts binary files into various hexadecimal formats used by in-circuit emulators and PROM programmers. **rel80** displays symbol table names, values and program section sizes, and may be used to examine object modules in various formats. See Chapter 8 for more information on the programming support utilities.

Assembly Language Source Code Format

x80 assembly language consists of lines of text in the form:

    <u>label:</u> <u>instruction mnemonic</u> <u>operand(s)</u> ; <u>comment</u>

or

    <u>label:</u> <u>assembler directive</u> <u>operand(s)</u> ; <u>comment</u>

where ':' indicates the end of a label and ';' defines the start of a comment. The end of the line terminates the comment. Since **x80** uses these separators, users are free to format each source line as they prefer.

Assembler directives may be of two types:

1) those which allocate storage, possibly initialized to a specified value, and cause the assembler to produce object code, and

2) those which control the behavior of the assembler, but do not directly produce object code.

Assembler directives are implemented independently of the target processor, and <u>are</u> <u>not</u> those defined by the manufacturer's documentation. Certain assembler directives allow the list of operands to continue on successive lines.

x80 instruction mnemonics and assembler directives may be written in upper or lower case; the following are all legal and equivalent:

```
    .byte   .BYTE   .bYtE
```

The C compiler generates lowercase assembly language.

Labels

A source line may begin with zero, one or more labels. Each label is a name terminated by a colon, e.g.:

```
    data1:  .byte 56H
```

A label may be viewed as a convenient way of allocating the current value of the program counter to the name. The label does not strictly refer to the statement on the same line of source code. Some assembler directives do not cause the assembler to generate object code and in such cases the label will have the value of the address of the next byte of code which would be generated for contiguous source code; it is also quite acceptable for a label to stand on a line by itself or with only a comment.

Labels are frequently used in conjunction with jump or call instructions and are also used for references to data. During pass one of the assembler, labels are placed in the symbol table together with their value.

Instruction Mnemonics

The processor instruction mnemonics for the Z80/HD64180 cross assembler are listed later in this chapter. Instruction mnemonics consist of a string of characters terminated by whitespace (space or tab character) or, if no operands are to follow, by the start of a comment.

Assembler Directives

Assembler directives start with the '.' character and are terminated by whitespace or, if no operands are to follow, by the start of a comment.

A quick reference description of each assembler directive appears later in this chapter.

Operands

Each instruction mnemonic or assembler directive can take zero or more operands. These are given in a list, starting after the whitespace which terminated the mnemonic or directive, usually separated by commas, and terminated by the end of line or the start of a comment.

Comments

Comments are ignored by the assembler. Comments begin with a semicolon and are terminated by the end of the line.

Output from the Assembler

x80 is capable of generating four different streams of output: object code, symbol table, program listing and error listing. Each of these output streams can be directed to a file or STDOUT by use of runtime options.

Object Code Output

The object code file produced by the assembler is a sequence of bytes for eventual execution on the target machine. This will normally consist of machine instructions, initialized data and, if a relocatable file is generated, relocation information. The object code file may also contain a symbol table, as well as a special debug symbol table.

The output code may be viewed as being in two parts: **(i)** skeleton object code, and **(ii)** relocation and symbol table information. The skeleton object code is generated by the assembler with temporary values where addresses are to be relocated. The relocation stream includes the **_text**, **_data** and **_debug** psects, and contains appropriate information for the relocation of address sized items.

This method is fairly efficient and allows linked executable objects to be relocated at runtime if required. The relocatable object format does, however, limit what the user can do when writing assembly language programs:

1) origins cannot be reset to a point before the current value of the program counter.

2) there are restrictions on expressions which involve relocatable or undefined terms. Such expressions are restricted to: label+constant, label-constant, constant+label, constant-label and (when in the same psect) label-label.

Symbol Table

The symbol table produced by the assembler is a list of names found during the first pass of the assembler together with the values that have been computed for them. External or undeclared objects are listed without values. The section to which these values belong is also included in the symbol table. The symbol table is produced at the end of pass one after any span dependencies have been resolved.

Listings

The listing stream contains the source code used as input to the assembler, together with the hexadecimal representation of the corresponding object code and the address for which it was generated. The listing stream can be switched off, sent to a file or sent to STDOUT.

The listing file is generated in pass two of the assembly; if the assembly does not reach this stage, no listing output will be generated. The contents of the listing stream depends on the occurrence of **.LIST** directives in the source. The format of the output is as follows:

<u><address></u> <u><generated code></u> <u><source code line></u>

where <u><address></u> means the hexadecimal address for which the <u><source code line></u> has been assembled, <u><generated code></u> means the hexadecimal representation of the object code generated by the assembler and <u><source code line></u> means the original source line input to the assembler. If expansion of data, macros and included files is not enabled, the <u><generated code></u> print will not contain a complete listing of all generated code.

Addresses in the listing output are the offsets from the start of the current section.

Error Messages

Error messages may be produced during pass one or pass two. They consist of a line number and a plain English message. If errors are detected during pass one, the assembler terminates at the end of the pass and no object code output or listing is generated. A list of the error messages that the assembler generates appears at the end of this chapter.

Internal Organization of x80

The first suite of routines splits the source text into tokens, i.e. names, numbers etc., and performs some error checking. This suite of routines includes those which save contexts when files are included. Note that macros are stored as a sequence of tokens which are re-input to pass one as needed. This imposes some limitations on macro-processing; they cannot, for example, be subsequently concatenated.

Pass one performs the bulk of the error detection. It produces two major outputs: a symbol table giving the value of each symbol encountered and an intermediate temporary file for input to pass two. Pass one handles all the expansions of **.DEFINE** and **.MACRO** directives and conditional assemblies.

The algorithm to optimize span dependent instructions is called at the end of pass one and it may alter values stored in the symbol table. The routine to print the symbol table is then called; this will also check for any missing identifiers.

Pass two reads the temporary file, and with the aid of the symbol table produces the correct object code file and any listing information. Some errors are reported during this pass: the major one is the phase error which is reported when symbols have a different value on pass one and pass two. Note that error messages of this kind may contain misleading line numbers.

Rules for Assembly Language Source

The source file consists of a number of lines of ASCII characters. The maximum line length of the source is 128 characters and the final character of a line should be a newline character (or sequence).

Use the conventions described below within each source line.

Identifier Names

Names used within a program may be of any length, but only the first 127 characters of a name are used for identification.

x80 is tolerant as to what it will accept as a valid name; basically anything that cannot be interpreted in any other way is acceptable. Thus, 2x8 and 2a8 would be accepted as names, but 2a8H is considered a hexadecimal number. Names are composed of letters, digits, underscore and dot characters in any order. Global **C** data objects produce names beginning with an underscore, e.g. **_end.** It is therefore advisable to adopt the convention of reserving such symbols for interfacing C with the assembler.

The base of the number is determined by a letter suffix:

B or b	binary
O, Q, o or q	octal
H or h	hexadecimal
D, d or none	decimal

Note that the UNIX and C convention that a leading 0 designates an octal number and a leading 0x designates a hexadecimal number is not supported by **x80.**

A number must begin with a digit (i.e. a character from 0 to 9); as an example the number 10 is shown in the various bases:

10	decimal (may also be written 10D)
1010B	binary
12Q	octal (Q is recommended rather than O)
0AH	hexadecimal (0 distinguishes from name AH)

Floating Point Numbers

x80 supports the format specified by ANSI/IEEE Std 754-1985, which is implemented by arithmetic chips such as the AMD9512. There are two acceptable lengths for floating point numbers:

32-bit single precision comprising
[Signbit][8-bit Exponent][23-bit Mantissa]

and

64-bit extended precision comprising
[Signbit][11-bit Exponent][52-bit Mantissa]

The sign bit is 0 for positive values and 1 for negative values. The exponent is biased by 127 (2^7-1) for single precision and by 1023 (2^10-1) for extended precision. The mantissa together with the sign bit represents a number in sign and magnitude notation. There is an implied 1 in the first bit of the mantissa: it it assumed to be normalized. The value represented is:

```
N = (-1)^S x 2^(E - bias) x (1.M)
```

The number may be entered in normal floating point format e.g. 1.01, or scientific format, e.g. 1.23E-3. Numbers must begin with a leading digit and contain a period; character strings beginning with a period are taken to be names.

x80 also supports the format used by the AMD9511 and the format used by the DEC PDP-11.

Characters

A common use of microprocessors is the transfer of characters between devices such as keyboards and printers. The assembler includes a convenient representation for both individual characters and strings of characters. The value of a character depends upon the alphabet in use. The assembler supports the American variant of the ISO 5 alphabet, commonly referred to as ASCII.

ASCII Character Set

An ASCII character is composed of two fields: a 7-bit field which distinguishes one character from another,[and an optional one-bit parity field which provides a check for transmission errors. The value of the parity bit is controlled by the **.ASCII** directive; its default value is 0. If required, however, it may be set to 1, or to a value which depends on whether there is an odd or even number of 1s in the binary representation of the character; see the description of the **.ASCII** directive later in this chapter for details.

ASCII Character Set

Octal	*Hex*	Char	*Octal*	*Hex*	Char	*Octal*	*Hex*	Char	*Octal*	*Hex*	Char
000	*00*	NUL	*040*	*20*	SP	*100*	*40*	@	*140*	*60*	`
001	*01*	SOH	*041*	*21*	!	*101*	*41*	A	*141*	*61*	a
002	*02*	STX	*042*	*22*	"	*102*	*42*	B	*142*	*62*	b
003	*03*	ETX	*043*	*23*	#	*103*	*43*	C	*143*	*63*	c
004	*04*	EOT	*044*	*24*	$	*104*	*44*	D	*144*	*64*	d
005	*05*	ENQ	*045*	*25*	%	*105*	*45*	E	*145*	*65*	e
006	*06*	ACK	*046*	*26*	&	*106*	*46*	F	*146*	*66*	f
007	*07*	BEL	*047*	*27*	'	*107*	*47*	G	*147*	*67*	g
010	*08*	BS	*050*	*28*	(	*110*	*48*	H	*150*	*68*	h
011	*09*	HT	*051*	*29*	)	*111*	*49*	I	*151*	*69*	i
012	*0A*	LF	*052*	*2A*	*	*112*	*4A*	J	*152*	*6A*	j
013	*0B*	VT	*053*	*2B*	+	*113*	*4B*	K	*153*	*6B*	k
014	*0C*	FF	*054*	*2C*	,	*114*	*4C*	L	*154*	*6C*	l
015	*0D*	CR	*055*	*2D*	-	*115*	*4D*	M	*155*	*6D*	m
016	*0E*	SO	*056*	*2E*	.	*116*	*4E*	N	*156*	*6E*	n
017	*0F*	SI	*057*	*2F*	/	*117*	*4F*	O	*157*	*6F*	o
020	*10*	DLE	*060*	*30*	0	*120*	*50*	P	*160*	*70*	p
021	*11*	DC1	*061*	*31*	1	*121*	*51*	Q	*161*	*71*	q
022	*12*	DC2	*062*	*32*	2	*122*	*52*	R	*162*	*72*	r
023	*13*	DC3	*063*	*33*	3	*123*	*53*	S	*163*	*73*	s
024	*14*	DC4	*064*	*34*	4	*124*	*54*	T	*164*	*74*	t
025	*15*	NAK	*065*	*35*	5	*125*	*55*	U	*165*	*75*	u
026	*16*	SYN	*066*	*36*	6	*126*	*56*	V	*166*	*76*	v
027	*17*	ETB	*067*	*37*	7	*127*	*57*	W	*167*	*77*	w
030	*18*	CAN	*070*	*38*	8	*130*	*58*	X	*170*	*78*	x
031	*19*	EM	*071*	*39*	9	*131*	*59*	Y	*171*	*79*	y
032	*1A*	SUB	*072*	*3A*	:	*132*	*5A*	Z	*172*	*7A*	z
033	*1B*	ESC	*073*	*3B*	;	*133*	*5B*	[	*173*	*7B*	{
034	*1C*	FS	*074*	*3C*	<	*134*	*5C*	\	*174*	*7C*	|
035	*1D*	GS	*075*	*3D*	=	*135*	*5D*	]	*175*	*7D*	}
036	*1E*	RS	*076*	*3E*	>	*136*	*5E*	^	*176*	*7E*	~
037	*1F*	US	*077*	*3F*	?	*137*	*5F*	_	*177*	*7F*	DEL

ASCII characters with octal codes 000-037 are non-printing. They are called control characters, and either perform cursor motion, such as Line Feed, (octal 12), or are mapped to various functions, such as ringing a BeLl (octal 007).

Note that they are cognates of the next column of characters (starting with @ and ending with _). Hence they may be considered control versions of printing characters. Such characters are represented to the assembler as two-character escape sequences, of which the first is ^ and the second is the associated printing character. Hence, the ASCII character with octal code 007, hex code 07, may be sent to the assembler as ^G, and when output will (usually) ring a bell, flash the screen, etc. Note that '^' merely represents itself, while '^^' represents the RS (Record Separator) control character.

Not all devices can handle the full character set. Some teleprinters, notably the venerable Teletype TTY33tm, its clones and relatives, can neither generate nor print characters with octal codes greater than 137. (They can, however, produce control characters). That means they cannot produce or emit lower-case letters. [Often, the systems to which such printers are attached map transmitted upper-case letters (octal codes 101-132) to their lower-case equivalents (octal codes 141-162). Upper-case letters are then signified with a slash preceding the character itself.]

Common control characters may also be represented using a similar notation to that used in C compilers, such as:

'\n'	LF	newline	(12Q)
'\r'	CR	return	(15Q)
'\t'	TAB	tab	(09Q)
'\b'	BS	backspace	(08Q)
'\e'	ESC	escape	(033Q)
'\i'	SI	shift in	(017Q)
'\o'	SO	shift out	(016Q)

A similar escape method, \nnn, is also supported (where nnn is an octal number). Thus, the BEL character could be written as '\007' and the DEL character as '\177'.

All printable characters may be represented as literals by enclosing them in quotes, such as: 'A', 'z', '*', and so forth. The single quote character itself is represented by '''.

Strings can be entered using the **.TEXT** directive. A string is written as a sequence of characters delimited by double quotes. The backslash '\' escape mechanism described above can be used within strings. The caret '^' escape convention is, however, not supported for strings. The parity bit cannot be explicitly set or reset when using the .TEXT directive; the **.ASCII** directive determines how the parity bit is calculated.

Numeric Labels

Numeric labels may be used as an alternative to name labels. Numeric labels are sometimes called local symbols, but to avoid confusion this term is not used here. Numeric labels are used for two main purposes:

1) to avoid the programmer having to generate a meaningful name for a label to which reference will only be made within a few lines of its definition, and

2) within macro expansions; if a macro is to be used more than once the macro body cannot define a named label because this label will then be defined every time the macro is used giving rise to assembly errors.

The region of text within which a numeric label may be used is known as the numeric label block.

The numeric label block is delimited by:

1) a name label, or

2) a macro expansion

There is no restriction on the value of the number used to form the label, but values above 32000 may cause unpredictable results. In practice only a small number of active numeric labels is needed.

A simple example of use of numeric labels is shown below:

```
                           .external   init, inch
                           .external    error, process
                       start:
0000   CD0000              call    inch
0003   1803                jr  1$
0005   CD0000              call    error
                       1$:
0008   CD0000              call    init
                       main:
000B   CD0000              call    inch
000E   1803                jr  1$
0010   CD0000              call    error
                       1$:
0013   CD0000              call    process
0016   CD0B00              call    main
                           .end
```

Expressions

Constants are required at many points within the program. They are frequently used, for example, as operands for machine instructions or assembler directives. The assembler allows an arbitrarily complex expression that may be evaluated at assembly time wherever a constant is required. Calculations of expressions and sub-expressions use variables whose size is equal to or larger than the size of objects used by the target machine. Expressions evaluate to unsigned quantities.

An expression consists of a number of terms connected by binary operators, such as + or *. Expressions are evaluated to 16-bit precision. Note that all operators have the same precedence and expressions are evaluated from left to right. Thus 1 + 2 * 3 will have the value 9, being evaluated as (1 + 2) * 3. Parentheses may be used to force an alternative order of evaluation and accordingly 1 + (2 * 3) evaluates to 7.

The terms within an expression may be:

1) numbers

2) user defined variables introduced via the **.SET** or **.DEFINE** directives or declared as labels

3) the '.' symbol, which represents the current value of the program counter.

Note that when '.' is used as the operand of an instruction, it has the value of the program counter immediately prior to generating code for the instruction.

The assembler is a two pass program and certain variables may not have their values fixed until the end of pass one. Variables whose values are undefined during pass one will be given an arbitrary value of 0 in any calculation involving them. Those variables whose value is not known by the start of pass two will be flagged as errors. Use of a variable that is not defined when used in an expression during pass one, is a common cause of phase errors.

Terms may also be prefixed by a unary operator; five such operators are provided:

+	unary plus.
-	unary minus.
~	NOT operator, which complements each bit.
.LBYTE	returns the low order 8 bits the value of an expression.
.UBYTE	returns the low order 8 bits the value of an expression after it has been shifted right by 8 places. This is functionally equivalent to returning the higher order byte when dealing with a 16-bit expression.

A comprehensive set of binary operators is provided:

+	addition
-	subtraction
*	multiplication
/	integer division (rounds down to the nearest integer)
%	remainder (e.g. **a % b** has the value **a - (a/b))**
&	bitwise intersection of two terms
|	bitwise union of two terms
^	bitwise exclusive OR (differ) of two terms
<<	shift left (e.g. **a << b** has the value of **a** shifted left **b** places)
>>	shift right (e.g. **a >> b** has the value of **a** shifted right **b** places)

When producing relocatable code, addresses may only be used in expressions when the result can be expressed as a regular relocatable quantity or as a constant. The following table details the validity of most common expressions.

valid	invalid
	.LBYTE label1
	.UBYTE label2
label1 + constant	label1 + label2
label1 - constant	label1 & label2
constant + label1	label1 * constant
constant - label1	label1 / constant
	label1 % constant
	label1 * label2
	label1 % label2
label1 - label2	label1 - label2

Note that label1 - label2 is only valid when both are in the same program section and the same module.

Logical operations on relocatable addresses are not allowed and will be flagged as relocation errors.

Processor Instruction Mnemonics

The processor instruction mnemonics for this assembler follow those defined in the manufacturer's documentation.

x80 recognizes the following processor instruction mnemonics:

adc	daa	inc	ldir	ret	rrd
add	dec	ind	neg	reti	rst
and	di	indr	nop	retn	sbc
bit	djnz	ini	or	rl	scf
call	ei	inir	otdr	rla	set
ccf	ex	jbr	otir	rlc	sla
cp	exx	jp	out	rlca	sra
cpd	halt	jr	outd	rld	srl
cpdr	im0	ld	outi	rr	sub
cpi	im1	ldd	pop	rra	xor
cpir	im2	lddr	push	rrc	
cpl	in	ldi	res	rrca	

For an exact description of the above instructions, refer to the manufacturer's documentation.

Extra Instruction Mnemonics

Mnemonics taking the form **jb*** are **x80** span-dependent extensions to the manufacturer's standard mnemonics.

The 64180 is one of various microprocessors that implement span-dependent instructions; the number of bytes required for an instruction depends on its operands. Examples are the jp and jr instructions; if the destination is within 128 bytes of the instruction, the optimal instruction to use is a **jr** instruction requiring two bytes. Otherwise a three byte jp instruction is required.

x80 includes an optimal algorithm to make these choices if the user specifies the instruction mnemonics as:

jbr for **jp** or jr

Assembler Directives

This section describes some of the directives included with the assembler in general terms. Specific descriptions about each directive appear in the section "x80 Assembler Directives" later in this chapter.

Several directives are concerned with the allocation of storage:

.BYTE allocates storage in 8-bit (byte) quanta

.WORD allocates storage in 16-bit (2 byte) quanta

.DOUBLE allocates storage in 32-bit (4 byte) quanta

.FLOAT allocates storage in 32-bit (4 byte) quanta

.LFLOAT allocates storage in 64-bit (8 byte) quanta

.TEXT indicates that a string of characters is to be stored

.ADDR indicates that addresses are to be stored; the mode of storage depends on the target machine, but will often be 16-bit.

A useful feature of the assembler is the ability to split a program into several sections. Thus the program code and fixed constants may be in one section which can be written into ROM, whereas the variable components of the program may be directed to another section which will be placed in RAM. This is achieved using the **.PSECT** directive.

A simple macro facility is incorporated in the assembler. Macros are defined using the **.MACRO** and **.ENDM** directives. Macro expansion takes place during pass one of the assembler, the expanded macro being written out to the temporary file ready for pass two. Macros may take up to nine operands.

A block of program bounded by a **.REPEAT** and **.ENDR** can be repeated n times, where n is defined by an expression. The repeat directive is effectively an unnamed macro performed n times.

It is possible to assemble sections of a program conditionally using the .IF, and **.ENDIF** directives.

The **.INCLUDE** directive allows other files, for example those containing standard macro expansions or data definitions, to be incorporated into the program.

The **.LIST** directive allows full control over which sections of the program will be listed. It is possible to control the printing of included files and macro expansions, and to exercise simple control over which sections will be listed. Both the code address and the assembled bytes are expressed in hexadecimal form.

Text Stream Formats

x80 produces three streams of text to aid the user:

1) program listing,

2) symbol table, and

3) error messages.

Each line in the program listing consists of the source code line number, the target address, and the generated code followed by the original source line. The line number is shown in decimal whereas the address and code are shown in hexadecimal. If errors are detected in the code, the error stream will contain: **(i)** the line number of each error together with an error message in English, and **(ii)** after all the error messages, the number of errors found in the source text. Each line of the symbol table contains the value of a name, the program section in which the symbol is to be found and the symbol name. These streams of text can either be sent to different files or can be interspersed. If interspersed, the symbol table will appear first, followed by the pass two listing. Pass two errors appear within the listing immediately adjacent to the appropriate line. If pass one errors are detected, no symbol table, listing or pass two errors will be produced.

Error Messages

Error messages are written in English and are normally sent to the error channel (STDERR); they may, however, be sent to a file by setting the appropriate options when running the cross assembler.

The form of an error message is:

```
    file        line number : error message
```

The line number may consist of several fields if the error occurred in an included file. For example:

```
    mycode      301 : 41 : invalid name
```

indicates that the error occurred in line 41 of the current file (named **mycode**) which was included at line 301 of the main source file.

Certain errors, such as a table overflow, are fatal and will cause the cross assembler to terminate.

Error messages are preceded by a line of the form:

x80 (n):

where n is the pass number reached.

A list of **x80** error messages appears at the end of this chapter.

The Symbol Table

A symbol table may be output at the end of pass one.

The symbols are output one per line in the format:

0xnnnn? name

where nnnn is the hexadecimal value and ? is the type chosen from one of the following set:

A	absolute
t	local name in _**text** psect
T	public name in _**data** psect
d	local name in _**data** psect
D	public name in _**data** psect
g	local name in _**debug** section
G	public name in _**debug** section
u/U	symbol not defined

The numeric labels (which are not necessarily unique) are distinguished from one another by a numeric suffix, which gives the position of the numeric label block for which the label is valid. The value of the suffix is n if the label is in the nth numeric label block.

Listings

A listing might take the form:

```
        Code    Assembled
Line    Address   Bytes    Source

  1                        .list +
  2                        .psect _data
  3
  4     000000  41         A1:    .byte   'A'
  5     000001  58         A2:    .byte   'X'
  6     000002  07         A3:    .byte   7
  7
  8                        .even
  9     000004  007B       B1:    .word   123
 10     000006  87654321 L1:      .double 087654321h
 11     00000A  3F9E192A          .float  1.234
 12     00000E  3FF3C319 LF1:     .lfloat 1.234
                C5A3E39F
 13     000016  6578616D T1:      .text   "example text",07
                706C6520
                74657874
                07
 14                        .end
```

Both the code address and the assembled bytes are expressed in hexadecimal form.

Invoking x80

You invoke **x80** via a command line of the form:

x80 <options> <file>

where <options> is zero or more of the valid command line options described below and <file> is an assembly language source file name.

The option **-help** will cause **x80** to output a usage reminder consisting of a list of valid command line options and their syntax.

If a option includes a value, then the kind of value it requires is indicated by one of the following codes. This code should be specified immediately after the option name:

Code	Kind of Value
*	character string
#	integer (size of host machine word)
##	integer (long, 32-bit)
?	single character

An integer is interpreted as hexadecimal if it begins with "0x" or "0X", or as octal if it begins with '0', and otherwise as decimal. A leading '+' or '-' is permitted.

If a option may be meaningfully specified more than once, then the value code is followed by '^'.

Command Line Options

x80 accepts the following command line options, each of which is described in detail below:

x80 -[c d*^ e f# g h* i* l +l n o* st# sd#
sz# s v x] <file>

-c produce cross-reference information. If a listing file is requested cross-reference information will be added at the end of the listing file; otherwise cross-reference information is sent to **STDOUT**.

-d* where * has the form name=#, define name to have the value specified by #. This option is equivalent to using a **.SET** directive of the form:

-e write assembly language error messages to the standard output channel STDOUT. The default is to write error messages to STDERR.

-f# select floating point format according to the value of #. Valid values for # are 0 for DEC PDP-11 style, 1 for AMD9511 style and 2 for ANSI/IEEE Standard 754-1985.

-g treat undefined symbols as external symbols. The default is to treat undefined symbols as errors.

-h* include the file specified by * before starting assembly. This option is useful for specifying header files that contain information that is to be used in several modules.

-i* change the prefix used with **.include <filename>** from the default (no prefix) to the string *. Multiple prefixes (including the null prefix) to be tried in order may be specified, separated by the character **|**.

+l write a listing to the standard output channel STDOUT. The default is not to output a listing.

-l write a listing to the listing file. The name of the listing file is determined by replacing the rightmost extension in the input file name with the "ls" extension. For example, if the input file name is **prog.s**, the listing file name is prog.ls. Similarly, if the input file name is **prog.old.v2**, the listing file name is **prog.old.ls**. However, your host system rules for file names should always be taken into account when you use this option.

-n add local symbols into the symbol table. The symbol table is produced when the **-s** option is specified.

-o* write object code to the file *. If no file name is specified, the output file name is derived from the input file name, by replacing the rightmost extension in the input file name with the character **'o'**. For example, if the input file name is **prog.s**, the default output file name is **prog.o**.

-st# set the _**text** section offset to the value specified by #. By default, the _**text** section offset is taken to be zero. This option is useful when writing small self-sufficient assembler programs; by using the **-st#** and **-sd#** options, you can specify the addresses where your program is to be run and save the expense of a link.

-sd# set the _**data** section offset to the value specified by #. By default, the _**data** section offset is taken to be zero. This option is useful when writing small self-sufficient assembler programs; by using the **-st#** and **-sd#** options, you can specify the addresses where your program is to be run and save the expense of a link.

-s create a symbol file. By default no symbol file is created. The name of the symbol file is derived from the name of the input filename by replacing its rightmost extension with **"sm"**. For example, if your input filename is **prog.s**, the symbol file name is **prog.sm**.

-v write to STDOUT the version and the release number of the cross assembler.

-x add debug symbol information to the symbol file if any is produced.

x80 Assembler Directives

This section consists of quick reference descriptions for each of the **x80** assembler directives. Each directive is documented according to the following conventions:

Strings shown in lower case are non-terminal symbols and normally refer to another definition to be found elsewhere. Words are sometimes concatenated using the visible space character '_' to make the object name more readable.

Strings shown in upper case are terminal symbols to be used in either upper or lower case when writing assembly language source.

The following symbols have special meaning: { } | = *

{ }	enclose a section that is optional
{ }*	enclose a section of text that is optional and can be repeated a number of times.
|	separate two or more items, one of which must appear.
=	the non-terminal symbol on the left of the equal sign is defined as consisting of the symbols on the right.

Other punctuation symbols such as the comma ',' and square brackets '[' are shown as you would enter them in the assembly language source.

NAME

.ADDR - allocate storage for addresses

SYNOPSIS

```
.ADDR elemlist
elemlist = xelem { , xelem }*
xelem    = initial_value
```

FUNCTION

This directive allocates, and optionally initializes, storage to hold addresses in the natural byte order of the target machine.

Most 8- and 16-bit machines have an address size of 16 bits and in such cases this directive is synonymous with **.WORD.** For machines with an address size greater than 16-bits this directive is synonymous with **.DOUBLE.** A list of addresses may be allocated with the items separated by commas.

Items specified using this directive may be relocated at link time.

EXAMPLE

```
routab:    .ADDR   main,    ; main routine
       dummy,
       setheap,    ; heap initialization
       dummy
```

SEE ALSO

.BYTE, .DOUBLE, .FLOAT, .LFLOAT, .WORD

NOTES

If the element list extends beyond a single line, the last active element on the non-terminal lines must be a comma separator, possibly followed by whitespace or comments.

NAME

.ASCII - modify parity bit

SYNOPSIS

.ASCII parity

FUNCTION

The only alphabet currently supported by the **x80** assembler is the American variant of the ISO 5 alphabet (ASCII). Characters in this alphabet consist of a 7-bit field plus a 1-bit parity field which occupies the most significant bit in an 8-bit byte. This bit may be always 0, always 1, or it may depend upon the number of 1's within the character.

The **.ASCII** directive allows four options:

7 sets the parity bit to 0 (default)

8 sets the parity bit to 1

.EVEN sets the parity bit such that there is always an even number of 1's in the character

.ODD sets the parity bit such that there is always an odd number of 1's in the character

EXAMPLE

```
.ASCII  7
.ASCII  8
.ASCII  .EVEN
.ASCII  .ODD
```

SEE ALSO

.TEXT

NAME

.BYTE - allocate an 8-bit sized variable space

SYNOPSIS

```
.BYTE elemlist
elemlist = xelem { , xelem }*
xelem    = initial_value
         | initial_value ( repeat_count )
         | [ size ]
```

FUNCTION

This directive allocates, and optionally initializes, storage for 8-bit sized variables. Initialization can be specified for each item by giving a series of values separated by commas or by using a repeat count. As with other directives which take a list of operands, successive items may be placed on separate lines. Note that the expression must not involve any relocation at link time. The **[size]** operand reserves uninitialized storage.

EXAMPLE

```
                        ; as a buffer, but don't initialize
TRATBL:             .BYTE '0','1','2','3','4','5','6',
                    '7','8','9','A','B',
                    'C','D','E','F' ; allocate 16 bytes
                        ; initialized to the values given
CHRBUF:             .BYTE ' ' (64) ; 64 bytes initialized to spaces
```

SEE ALSO

.ADDR, .DOUBLE, .FLOAT, .LFLOAT, .WORD

NOTES

If the element list extends beyond a single line, the last active element on the non-terminal lines must be a comma separator, optionally followed by whitespace or comments.

NAME

.DEFINE - give a permanent value to a symbol

SYNOPSIS

```
.DEFINE symbol = n_r_expr { , symbol = n_r_expr}*
n_r_expr = non_relocatable_expression
```

FUNCTION

This directive is used to associate a value with a symbol. Symbols declared with the **.DEFINE** directive may not subsequently have their value altered; the **.SET** directive should be used if this is necessary. The expression to the right of the equal sign must be fully defined at the time the **.DEFINE** directive is assembled.

EXAMPLE

```
.DEFINE        FALSE = 0,    ; initialize these values
               TRUE = -1

.DEFINE        TABLEN = TABFIN - TABSTA ; compute table length

.DEFINE        NUL = 00H,    ; define strings for ASCII characters
               SOH = 01H,
               STX = 02H,
               ETX = 03H,
               EOT = 04H,
               ENQ = 05H
```

SEE ALSO

.SET

NOTES

It is easy to forget the comma separators.

NAME

.DOUBLE - allocate a double word four byte sized variable area

SYNOPSIS

```
.DOUBLE elemlist
elemlist = xelem { , xelem }*
xelem    = initial_value
         | initial_value ( repeat_count )
         | [ size ]
```

FUNCTION

This directive allocates, and optionally initializes, double word or four byte sized integer variables in the natural byte order of the target machine.

Initialization can be specified for each item by giving a series of values separated by commas or by using a repeat count. The initial value must be in integer format for correct initialization. As with other directives which take a list of operands, successive items may be placed on separate lines. The **[size]** operand reserves uninitialized storage.

Initial values must not involve any relocation at link time, except where the address size of the machine is 32-bits.

EXAMPLE

```
DUBaRR: .DOUBLE [10]    ; Allocate 10 items to form an
            ; array of uninitialized doubles

DONUM:  .DOUBLE 13768   ; initialize donum to a double

TENTAB: .DOUBLE 1,  ; powers of ten
            10,
            100,
            1000,
            10000,
            100000
```

SEE ALSO

.ADDR, .BYTE, .FLOAT, .LFLOAT, .WORD

NOTES

If the element list extends beyond a single line, the last active element on the non-terminal lines must be a comma separator, optionally followed by whitespace or comments.

NAME

.ELSE - conditionally assemble code sections

SYNOPSIS

```
.IF conditional_expression
    statements
    { .ELSE
    statements }
.ENDIF
```

FUNCTION

The **.IF**, **.ELSE** and **.ENDIF** directives allow conditional assembly. During pass one the conditional expression operand to the **.IF** directive is evaluated. If the conditional statement evaluates to TRUE, the statements following it are assembled up to an **.ENDIF** or **.ELSE** directive. If the conditional expression evaluates to FALSE, the statements following it up to an **.ENDIF** or **.ELSE** directive are ignored.

If the **.IF** statement has an **.ELSE** part and if the conditional expression evaluates to FALSE, the statements between **.ELSE** and **.ENDIF** are assembled. Because the **.ELSE** directive is optional and in order to avoid the dangling else problem, all **.ELSE** directives are associated with the previous **.IF**.

EXAMPLE

```
.IF Extra_memory        ; check if extra memory
           ; is in machine build
   .DEFINE STACK_START = 4000H,
       STACK_LEN = 1000H
.ELSE
   .DEFINE START_START = 3800H,
       STACK_LEN = 800H
.ENDIF
```

SEE ALSO

.ENDIF, .IF

NAME

.END - stop the assembly

SYNOPSIS

```
.END { label }
```

FUNCTION

This directive stops the assembly process; any statements following it are listed but not assembled. The **.END** directive may be followed by an optional expression which is the entry point to the program. The operand is useful only for absolute assemblies which generate Intel hexadecimal directly; the information is lost when generating other output formats.

EXAMPLE

```
.END     PROG_INIT ; end assembly and cause code
         ; (when run) to begin at PROG_INIT
```

NAME

.ENDIF - end conditional assembly of code sections

SYNOPSIS

```
.IF conditional_expression
    statements
    { .ELSE
     statements }
.ENDIF
```

FUNCTION

The **.IF**, **.ELSE** and **.ENDIF** directives allow conditional assembly. During pass one the conditional expression operand to the **.IF** directive is evaluated. If the conditional statement evaluates to TRUE, the statements following it are assembled up to an **.ENDIF** or **.ELSE** directive. If the conditional expression evaluates to FALSE, the statements following it up to an **.ENDIF** or **.ELSE** directive are ignored.

If the **.IF** statement has an **.ELSE** part and if the conditional expression evaluates to FALSE, the statements between **.ELSE** and **.ENDIF** are assembled. Because the **.ELSE** directive is optional and in order to avoid the dangling else problem, all **.ELSE** directives are associated with the previous **.IF**.

EXAMPLE

```
.IF Extra_memory    ; check if extra memory is
       ; in machine build
   .DEFINE STACK_START = 4000H,
       STACK_LEN = 1000H
.ELSE
   .DEFINE START_START = 3800H,
       STACK_LEN = 800H
.ENDIF
```

SEE ALSO

.ELSE, .IF

NAME

.ENDM – end macro definition

SYNOPSIS

```
.MACRO name
   macro body
.ENDM
```

FUNCTION

This directive is used to terminate macro definitions.

EXAMPLE

```
; define a macro that places the length of
; a string in a byte prior to the string

.MACRO LTEXT
   .byte   2$-1$
1$:
   .text   ?1  ; text given as first operand
2$:
.ENDM
```

SEE ALSO

.ENDR, .EXITM, .MACRO, .REPEAT

NAME

.ENDR - end a repeat section

SYNOPSIS

```
.REPEAT [expression]
   macro_body
.ENDR
```

FUNCTION

The **.REPEAT** directive is used to cause the assembler to re-read a section of source text a number of times. The repeat section is terminated by an **.ENDR** directive.

EXAMPLE

```
; Shift a value n times
.MACRO ASLN
   .REPEAT [?1]
       asl
   .ENDR
.ENDM
 ; Use of above macro
ASLN 5
```

SEE ALSO

.ENDR, .MACRO

NAME

.EVEN - assemble next byte to an even address

SYNOPSIS

```
.EVEN
```

FUNCTION

This directive places the next assembled byte at an even address. It is needed by those processors which have a 16-bit instruction, but address memory in 8-bit bytes (e.g. TMS9900).

EXAMPLE

```
VOWTAB: .BYTE   'A','E','I','O','U'
        .EVEN   ; Ensure aligned at even address
TENTAB: .WORD   1,
                10,
                100,
                1000
```

NAME

.EXITM – terminate a macro definition

SYNOPSIS

```
.MACRO name
   macro body
   conditional directive
       .EXITM
   continuation of macro body
.ENDM
```

FUNCTION

This directive is used to exit from a macro definition before the **.ENDM** definition is reached. The directive is usually placed after a conditional assembly directive.

EXAMPLE

```
.macro ctrace
   .if tflag = 0
       .exitm
   .endif
   jsr ?1
.endm
```

To use macro ctrace, call it with the name of the desired routine, e.g. ctrace display

SEE ALSO

.ENDM, .ENDR, .MACRO, .REPEAT

NAME

.EXTERNAL - declare symbol as being defined elsewhere

SYNOPSIS

```
.EXTERNAL { psect } symbol { , symbol }*
```

FUNCTION

Visibilty of symbols between modules is controlled by the **.EXTERNAL** and **.PUBLIC** directives. Symbols which are defined in other modules must be declared as external. So that standard headers may be used within programs, a symbol may be declared to be both external and public in the same module.

To allow compatibility with the conventions of the C programming language, when a symbol is declared as external, it is treated as though there is also a public definition.

If no psect is given, the current psect is taken by default.

The directive is only appropriate when generating relocatable code, because absolute code segments cannot be linked together.

EXAMPLE

```
.EXTERNAL  OTHERPROG
.EXTERNAL  _data TABLE ; This table is in data segment
.EXTERNAL  _text SQRT  ; This routine is in text segment
```

SEE ALSO

.PUBLIC

NAME

.FLOAT – allocate a floating point four byte sized variable area

SYNOPSIS

```
.FLOAT elemlist
elemlist = xelem { , xelem }*
xelem    = initial_value
         | initial_value ( repeat_count )
         | [ size ]
```

FUNCTION

This directive is used to declare floating point four byte sized variables and optionally to initialize the variables. Initialization can be specified for each item by giving a series of values separated by commas or by using a repeat count. As with other directives which take a list of operands, successive items may be placed on separate lines.

Note that the expression must not involve any relocation at link time. Integer initial values will be widened to floating point format.

The **.FLOAT** directive only reserves space and initializes areas. The programmer must ensure that the variable is used consistently with its declaration at runtime. Choice of a floating point format does not cause automatic loading of library routines which may be needed to handle floating point calculations at runtime. The format of the initialized number depends upon the setting of the **-fp** option.

EXAMPLE

```
FLOARR:    .FLOAT [10] ; allocate 10 items to form
                 ; an array of uninitialized floats

PI:  .FLOAT    3.1415926536 ; initialize a FLOAT
           ; labeled PI

FNARR:     .FLOAT 1.0,
           2.0,
           3.0
```

SEE ALSO

.ADDR, .BYTE, .DOUBLE, .LFLOAT, .WORD

NOTES

If the element list extends beyond a single line, the last active element on the non-terminal lines must be a comma separator, possibly followed by whitespace or comments.

NAME

.IF – conditionally assemble code sections

SYNOPSIS

```
.IF conditional_expression
    statements
    { .ELSE
     statements }
.ENDIF
```

FUNCTION

The **.IF**, **.ELSE** and **.ENDIF** directives allow conditional assembly. During pass one the conditional expression operand to the **.IF** directive is evaluated. If the conditional statement evaluates to TRUE, the statements following it are assembled up to an **.ENDIF** or **.ELSE** directive. If the conditional expression evaluates to FALSE, the statements following it up to an **.ENDIF** or **.ELSE** directive are ignored.

If the **.IF** statement has an **.ELSE** part and if the conditional expression evaluates to FALSE, the statements between **.ELSE** and **.ENDIF** are assembled. Because the **.ELSE** directive is optional and in order to avoid the dangling else problem, all **.ELSE** directives are associated with the previous **.IF**.

The conditional expression may take one of three forms:

1) an expression which is tested for non-zero. Thus:

   .IF ee

   is equivalent to

   .IF ee <> 0.

2) two expressions tested according to a relational operator. This conditional expression takes the form:

   <ins>&lt;expression&gt;</ins> <ins>&lt;relational operator&gt;</ins> <ins>&lt;expression&gt;</ins>

   where <ins>&lt;relational operator&gt;</ins> can be:

=	test for equality
<>	test for inequality
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

3) an expression tested according to a specified test operator. This conditional expression takes the form:

<expression> <test operator>
where the <test operator> can be:

.EVEN	true if at an even boundary

and

.ODD	true if at an odd boundary

Conditional directives may be nested to a depth of ten.

EXAMPLE

```
.IF Extra_memory
; Check if extra memory is in machine build
   .DEFINE STACK_START = 4000H,
       STACK_LEN = 1000H
.ELSE
   .DEFINE START_START = 3800H,
       STACK_LEN = 800H
.ENDIF

; Macro to allocate table space, sized either by
; second operand for a default of 64.
; The first operand is the name of the table.

.MACRO TABLE
   .if ?0 = 1       ; Only 1 operand given to macro?
      ?1 : .byte 0 (64)    ; Allocate 64 bytes if default
   .else
      ?1 : .byte 0 (?2)    ; Specify size of table
   .endif
.ENDM
```

SEE ALSO

.ELSE, .ENDIF

NAME

.INCLUDE - include text from another text file

SYNOPSIS

```
.INCLUDE "filename"
```

FUNCTION

This directive causes the assembler to take its input from the specified file until end of file is reached, at which point the assembler continues to read input from the line following the **.INCLUDE** directive in the current file. The directive is followed by a string which gives the name of the file to be included. This string must match exactly the name and extension of the file to be included, giving due regard to upper/lower case characters where the operating system recognizes such distinctions.

Included files often contain information that is required to be constant between a large number of modules, for example port addresses, character set definitions, specialist macros.

Files may be nested to a depth of ten, allowing an included file to include other files. Consistent error reporting will only occur for up to 16 included files.

EXAMPLE

```
.INCLUDE    "DATSTR"     ; use data structure library
.INCLUDE    "BLDSTD"     ; use current build standard
.INCLUDE    "MATMAC"     ; use maths macros
.INCLUDE    "PORTS82"    ; use ports definition
```

NAME

.LFLOAT - allocate a long floating point eight byte sized variable

SYNOPSIS

```
.LFLOAT elemlist
elemlist = xelem { , xelem }*
xelem    = initial_value
         | initial_value ( repeat_count )
         | [ size ]
```

FUNCTION

This directive allocates, and optionally initializes, long floating point eight byte sized variables.

Initialization can be specified for each item by giving a series of values separated by commas or by using a repeat count. Initial integer values will be widened to long floating point format. As with other directives which take a list of operands, successive items may be placed on separate lines. The **[size]** operand reserves uninitialized storage.

The initial value must not involve any relocation at link time.

The **.LFLOAT** directive only reserves space and initializes areas. The programmer must ensure that the variable is used consistently with its declaration at runtime. Choice of a floating point format does not cause automatic loading of library routines which may be needed to handle floating point calculations at runtime. The format of the initialized data depends upon the setting of the **-fp** option.

EXAMPLE

```
LFLARR:   .LFLOAT   [10]
          ; allocate 10 items
          ; to form an array of
          ; uninitialized lfloats
PI:  .LFLOAT        3.1415926536
          ; initialize lfloat labeled PI
```

SEE ALSO

.ADDR, .BYTE, .FLOAT, .DOUBLE, .WORD

NOTES

If the element list extends beyond a single line, the last active element on the non-terminal lines must be a comma separator, possibly followed by whitespace or comments.

NAME

.LIST - select sections to be listed

SYNOPSIS

```
.LIST switchlist | + | -
switchlist = setting part { setting part }*
setting    = + | -
part       = .IF | .INCLUDE | .MACRO | .TEXT
```

FUNCTION

This directive controls the parts of the program which will be written to the listing file. The command line used to invoke the cross-assembler must have a listing flag set for this directive to have any effect.

The parts of the program to be listed are specified by giving the following operands to the **.LIST** directive:

.IF the program lines which are not assembled as a consequence of **.IF, .ELSE** and **.ENDIF** directives

.INCLUDE included files

.MACRO macro expansions of code (not expanded text)

.TEXT multiple lines of code from **.BYTE, .TEXT, .WORD** directives and similar

Listings may be enabled using <u>+part</u> or suppressed by <u>-part.</u> All listing can be enabled by an operand of '<u>+</u>' only, and all listing disabled by an operand of '<u>-</u>' only. The initial setting is to allow listing of the main body and of text strings only.

EXAMPLE

```
.list  +.macro ; expand macros and
       ; repeated text

.repeat    10
.byte  1,2,4,8,16
.endr
```

NAME

.MACRO - define a macro

SYNOPSIS

```
.MACRO name
   macro_body
.ENDM
```

FUNCTION

This directive is used to define a macro. The name may be any previously unused name, a name already used as a macro, or an instruction mnemonic for the microprocessor. Macro names are not entered into the symbol table until **.ENDM** is encountered. Thus, it is possible to redefine an instruction as well as a macro.

Macros are expanded when the name of a previously defined macro is encountered. Operands, where given, follow the name and are separated from each other by commas.

The macro_body consists of a sequence of tokens not including the directives **.MACRO** or **.ENDM.** It may contain macro variables which will be replaced, when the macro is expanded, by the corresponding operands following the macro invocation. These macro variables take the form **?1** to **?9** to denote the first to ninth operands respectively. In addition, the macro variable **?0** contains the number of actual operands given when the macro was invoked.

If more than nine operands are given, **?9** contains the ninth operand and the rest of the line. Because **?0** contains the actual number of operands given, it is possible, though not straightforward, to handle more than nine operands.

A macro expansion may be terminated early by using the directive which, when encountered, acts as though the end of the macro has been reached.

Because macros are expanded in the first pass, the expression **?n** (where n is a digit) is recognized as a separate token; thus a label such as

?1T:
will be treated as a syntax error, although the use of a macro variable alone as a label, for example

?1:
is valid.

Macros may be nested four deep. Macros and repeat directives may be nested to a combined depth of ten.

EXAMPLE

```
; REDEFINE THE NOP INSTRUCTION

 .MACRO NOP
    halt
    nop
 .ENDM  ; each nop instruction now is preceded by a halt

 ; define a macro that places the length of a string
 ; in a byte prior to the string

 .MACRO .LTEXT
    .byte   2$-1$
1$:
    .text   ?1  ; text given as first operand
2$:
.ENDM


; MACRO TO DEFINE AN ELEMENT OF A COMPLEX DATA STRUCTURE

macro  el       ; define an element
.define    ?1 = elno    ; associate name with element # (state)
.byte  ?2       ; character for testing
.byte  ?3       ; next state if true
.byte  ?4       ; next state if false
.addr  ?5       ; address of associated routine
.byte  ?6       ; flags associated with this element
.set   elno = elno+1    ; increment element number
.endm

; specify flag values
.define F.ST = 1 << 0, ; stop if true
F.SF = 1 << 1,      ; stop if false
F.AP = 1 << 2,      ; action routine prior to test
F.AT = 1 << 3,      ; action routine if test true
F.AF = 1 << 4       ; action routine if test false

.define LF = 0AH,       ; line feed
CR = 0DH            ; carriage return

.set   elno = 0             ; initialize element number

.external  SENDNL, ; put a newline in character stream
   EATCH       ; remove the character from stream

; build data structure (using above element definition)
; this data structure can be used to modify streams
; of CR and LF to a single NL sequence.

el INI,    CR, BCR,    ALF,    SENDNL, F.ST + F.AT
el ALF,    CR, BCR,    INI,    SENDNL, F.ST + F.SF + F.AT
el BCR,    CR, BCR,    BLF,    EATCH,  F.ST + F.AT
el BLF,    LF, BCR,    INI,    EATCH,  F.ST + F.SF + F.AT
```

SEE ALSO

.ENDM, .EXITM, .REPEAT

NAME

.PAGE - start a new page in the listing file

SYNOPSIS

```
.PAGE { "char_string" }
```

FUNCTION

The **.PAGE** directive causes a formfeed to be inserted in the listing output.

If a string operand char string is specified, it will be used to generate the header for the next page only. If no string operand is specified, the page header will be the default set during pass one by the **.TITLE** directive.

EXAMPLE

```
.EXTERNAL  MULT, DIV
.PAGE "Local variables"
.BYTE  CHARIN, CHAROUT
.WORD  A, B, SUM
```

SEE ALSO

.TITLE

NAME

.PSECT - place code in a program section

SYNOPSIS

```
.PSECT psect_name
```

FUNCTION

This directive controls the allocation of code to program sections. The directive indicates that the following program code is to be assembled into a named psect to follow on from the last code entered into this named psect. The C cross compiler emits code for the three psects **_text, _data** and **_bss.**

Four general psects; **_text, _data, _bss,** and **_debug** are supported.

Note: the **_debug** section is for debugging purposes only and should not be used for any other purpose.

EXAMPLE

When writing assembly modules for linking with the compiler output, code could be directed to the appropriate psect as follows:

```
    .PSECT  _text
r1: .BYTE   ......
    .PSECT  _data
r2: .BYTE   ......
    .PSECT  _text
r3: .BYTE   ......
    .PSECT  _data
r4: .BYTE   ......
```

This will place **r1** and then **r3** into consecutive locations in **_text** and **r2** and **r4** in consecutive locations in **_data.**

NAME

.PUBLIC - declare a variable to be visible

SYNOPSIS

```
.PUBLIC identifier_list
identifier_list = identifier { , identifier }*
```

FUNCTION

Visibility of symbols between modules is controlled by the **.PUBLIC** and **.EXTERNAL** directives. A symbol may only be declared as public in one module. In order that standard headers may be used within programs, a symbol may be declared to be both public and external in the same module.

To allow compatibility with the conventions of the C programming language, when a symbol is declared as external, it is also treated as though there is a public definition.

Note that the directives **.PUBLIC** and **.EXTERNAL** are only appropriate when generating relocatable code, because absolute modules cannot be linked together.

EXAMPLE

```
.PUBLIC    SQRT    ; allow SQRT to be called
       ; from another module

SQRT:  ; routine to return a square root
       ; of a number >= zero
   .....
   .....
```

SEE ALSO

.EXTERNAL

NAME

.REPEAT - reread the following text a number of times

SYNOPSIS

```
.REPEAT [expression]
   macro_body
.ENDR
```

FUNCTION

The **.REPEAT** directive is used to cause the assembler to reread the following section of source text up to the next **.ENDR** directive. The number of times the source text will be re-read is specified by the expression operand. The **.REPEAT** directive is equivalent to a macro definition followed by the same number of calls on that macro.

EXAMPLE

```
; Shift a value n times
.MACRO ASLN
    .REPEAT [?1]
        asl
    .ENDR
.ENDM

; Use of above macro
ASLN 5
```

SEE ALSO

.ENDR, .MACRO

NAME

.SET - give a resetable value to a symbol

SYNOPSIS

```
.SET symbol = n_r_expr { , symbol = n_r_expr }*
n_r_expr = non_relocatable_expression
```

FUNCTION

This directive allows a value to be associated with a symbol. Symbols declared with **.SET** may be given a new value by a subsequent **.SET.** If it is required to prevent resetting, the **.DEFINE** directive should be used. The expression to the right of the equal sign must be fully defined at the time the **.DEFINE** directive is assembled.

EXAMPLE

```
; Initialize a linked list structure
; in contiguous space
; with the first element of each
; pointing to the next element

.define         no_of_elems = 10 ; these values will not alter
.define         elem_size = 10

.set            elemno = 1 ; initialize this changeable value

.repeat         no_of_elements
    .addr 1$    ; address of next element
    .word elemno ; label the element with a number
    .byte [elem_size] ; reserve space (uninitialized)
1$: ; this is the address of the next link

    .set  elemno = 1 + elemno ; modify element number
.endr

.end            ; note: the last link of the chain
    ; should be marked as "end of chain"
```

SEE ALSO

.DEFINE

NAME

.**TEXT** - place text into memory

SYNOPSIS

```
.TEXT  textitem { , testitem }*
textitem = string
         | non_relocatable_expression
```

FUNCTION

The **.TEXT** directive is used to place sequences of characters within the code output by the cross assembler. The storage that is allocated is initialized to the value of the operands given to this directive. The operands are decoded and then placed in consecutive bytes of memory, the size of the area being the number of bytes into which the operands are to be decoded.

A string is written as a sequence of characters delimited by double quotes; single quotes are used to allow single characters to be entered. Characters may be expressed as octal numbers using the backslash '\' as an escape character; if the backslash character is to be entered, the sequence "\\" may be used. Non-printing and control characters may be inserted by using the expression option. All characters are masked to a value in the range 0 .. 127 and the parity bit is calculated according to the option selected by the **.ASCII** directive.

The **.TEXT** directive initializes an area of memory with a sequence of bytes equivalent to the source operands. No length information or string termination sequence is generated, allowing the programmer full freedom of choice for conventions of this kind.

EXAMPLE

```
.TEXT "bell",'^G' ; bell followed by ringing of bell
.TEXT "bell\07" ; another form of above
.TEXT "bell", '\07' ; yet another form of above
.TEXT 62H, 65H, 6CH, 6CH, 07H   ; so are these!
```

SEE ALSO

.ASCII

NAME

.TITLE - define default header

SYNOPSIS

```
.TITLE "name"
```

FUNCTION

The **.TITLE** directive is used to set the default page header used when a new page is written to the listing output. If the **.PAGE** directive has a string operand, it will be used instead of the default on that occasion.

EXAMPLE

```
.TITLE "Project X special function library"
```

SEE ALSO

.PAGE

NAME

.WORD - allocate storage for a word two byte sized variable

SYNOPSIS

```
.WORD elemlist
elemlist = xelem { , xelem }*
xelem    = initial_value
         | initial_value ( repeat_count )
         | [ size ]
```

FUNCTION

This directive allocates, and optionally initializes, single word or two byte sized integer variables in the natural byte order of the target machine.

Initialization can be specified for each item by giving a series of values separated by commas or by using a repeat count. The initial value must be in integer format for correct initialization. As with other directives which take a list of operands, successive items may be placed on separate lines. The **[size]** operand reserves uninitialized storage.

Initial values must not involve any relocation at link time, except where the address size of the machine is 16-bits.

EXAMPLE

```
WARRAY: .WORD  [10] ; reserve 10 uninitialized
               ; word spaces

TENTAB: .WORD  1, ; power of ten table
     10,
     100,
     1000,
     10000

W1ARR: .WORD   1 (10) ; initialize ten words to value 1
```

SEE ALSO

.ADDR, .BYTE, .DOUBLE, .FLOAT, .LFLOAT, .TEXT

NOTES

If the element list extends beyond a single line, the last active element on the non-terminal lines must be a comma separator, possibly followed by whitespace or comments.

Assembler Error Messages

The following error messages may be generated by the assembler. Note that the assembler's input is machine-generated code from the compiler. Hence, it is usually impossible to fix things 'on the fly.' The problem must be corrected in the source, and the offending program(s) recompiled.

' missing

The quote character is missing.

attempt to redefine <name>

<name> has been redefined.

branch out of range

A branch target address is not within [-128,127] from the PC value.

can't create <file>

<file> could not be created. Check the filename against your host operating system's requirements. Check also for free space on your disk, and that you do not have too many files in your directory. Lastly, check that the directory permission bits (if applicable) are correct.

can't include <file>

The file specified in a .include directive could not be found. Check for spelling errors, pathnames and access privileges.

can't move origin backwards

It is not possible to move the program counter location backwards.

can't open input file <filename>

<filename> could not be found. Possible reasons include misspellings, wrong directory listings, accidental file deletion, or incompatible access rights.

can't read temp file (flush_tbuf)

The assembler has reached an inconsistent internal state. See "Symbol Error Too Many Reads" for instructions on reporting this error to customer support.

can't read temp file (get_tbuf)

The assembler has reached an inconsistent internal state. See "Symbol Error Too Many Reads" for instructions on reporting this error to customer support.

can't write temp file

An error occurred when trying to write in a temporary file. This problem does not arise in the context of creating a file, but when trying to write in it. The most likely reason is that the disk is full.

can't write temp file (flush_tbuf)

The assembler has reached an inconsistent internal state. See "Symbol Error Too Many Reads" for instructions on reporting this error to customer support.

---

constant expected

The cross assembler expected a constant term in a location where a variable was specified. This error is most likely to happen when a directive is evaluated. Check the syntax for that directive.

---

constant expected; .define can't use relocatable

The .define directive is restricted to constants.

.end expected

The directive **.end** is not present in the input file.

---

field error

The value specified does not lie in the authorized range for that symbol. Example; specifying a value greater than 256 for a byte variable.

---

file name expected for .include

.include takes a filename argument. None was provided.

---

if depth exceeded

Conditional directives may be nested to a maximum depth of ten (10).

---

invalid address

The assembler encountered an invalid address mode. Check the permissible addressing modes.

---

include depth exceeded

.includes may be nested to a maximum depth of ten (10). Check the logic of file inclusion.

---

invalid \n char

An unexpected carriage return was encountered.

---

invalid char

An invalid character appears in a name.

---

invalid index register

Self explanatory.

---

invalid .list mode

The .list directive was specified with a bad mode.

---

invalid macro name

Check the syntax for writing macro names.

---

invalid register

An incorrect register was specified for the desired opcode or addressing mode.

---

macro depth exceeded

Macros may only be nested four (4) deep.

name list overflow

The input program has too many identifiers. It should be rewritten with a smaller number of symbols, or split into two or more files which can be assembled separately.

<name> not defined

Symbol <name> was used, but no definition was found.

no input file

No input file was specified, or the specified input file did not contain a **.end** directive.

non terminated string

The final " is missing in a string specification.

numeric label already defined <numlab>

The numeric label <numlab> is already defined within the same context. Scopes for numeric labels are delimited by name labels or macro expansions.

offset too large

The offset specified is too large to fit in a byte. Most likely to happen in conjunction with use of the zero page (or direct addressing).

phase error < ... >

The most likely cause of this error is that variables have been declared in one section, then used as though they belonged to a different section. Check that all zero-page symbols ar properly defined.

psect name required

.psect requires one of the following valid section names:

_text	**_data**
_bss	**_debug**

relocation error (adding two relative terms)

The cross assembler does not support relocation commands involving addition of two relative terms. This error occurs when the ± operator is used incorrectly in a relocatable expression.

relocation error in expression

An invalid operator was used in a relocatable expression.

relocation error (subtracting two relative terms)

The cross assembler does not support relocation commands involving subtraction of two relative terms. This error occurs when the ± operator is used incorrectly in a relocatable expression.

repeat depth exceeded

The **.repeat** and **.macro** directives may be nested to a combined depth of ten (10).

symbol error too many reads

This error is part of a consistency check internal to the cross assembler. It should never happen. If it does appear, users are requested to copy all the files involved and report the problem (with appropriate documents) to customer support.

---

system error no free space

All dynamic space has been exhausted. This error indicates the input program is too large.

---

syntax error invalid identifier <name>

<name> is an invalid identifier. Check the syntax for identifiers.

---

syntax error <operator> invalid

<operator> was found invalid in the context where it appears. Check the rules for that operator. The program should be split into two or more files and assembled separately.

---

syntax error <symbol> unexpected

Self explanatory.

---

syntax error <token> expected

A token was expected, but not found.

---

syntax error <token> invalid in expression

<token> was used incorrectly in an expression. Check the rules for expression syntax.

---

too many input files

Currently, the cross assembler can only handle 15 input files at a time. This error will appear if more than 15 input files have been passed as arguments to the assembler. In this case, the first 15 files will be assembled correctly, after which this message will appear.

---

too many macros

The current cross assembler supports a maximum of 64 macros.

---

unknown opcode <opcode>

Self explanatory.

---

unsupported processor

The string specified to the **.processor** directive doesn't match the syntax for identifiers.

---

unsupported processor type <name>

The processor name specified in a **.processor** directive is wrong. Specifically, it refers to an unsupported processor type, since reference to supported processors different from the actual target processor will simply produce inapplicable assembly code.

---

wrong type for .public

The .public directive received a bad name or token. Check the rules for writing identifiers.