Table of Contents

ASSEMBLER

Introduction

This section is an overview of the 2500 A.D. 64180 Cross Assembler. The intent of this manual is to describe the operation of the Assembler. It is assumed that the user is familiar with the 64180 operation and instruction set.

The 2500 A.D. 64180 Assembler enables the user to write programs which can then be assembled into relocatable object code and linked to the desired execution address using the 2500 A.D. Linker.

The Assembler will process any size file, as long as enough memory is available. All the buffers used by the Assembler are requested and expanded as needed, with the exception of the Source Code Input Buffer, the Object Code Output Buffer and the Listing Buffer.

The Conditional Assembly section enables the user to direct the Assembler to process different sections of the source file depending on the outcome of assembly time operations. Conditionals may be nested to 248 levels, and the Assembler aids the programmer in detecting conditional nesting errors by not only checking for unbalanced conditional levels, but also by displaying the current active conditional level in the object code field of the listing.

The Assembly Time Calculation section will perform calculations with up to 16 pending operands, using 80 bit arithmetic. The algebraic hierarchy may be changed through the use of parenthesis.

The Listing Control section provides for listing all or just sections of the program, with convenient Assembler error detection overrides, along with Assembly Run Time Commands that may be used to dynamically change the listing mode. Also, in this section is a description of the LINKLIST directive which allows the linker to relocate listings

The 2500 A.D. Linker allows files to either be linked together or just used for external reference resolution. As with the Assembler, all buffers used by the Linker are requested as needed. The Linker is capable of outputting several different formats. The format may be changed by using an Assembler Directive or selecting the desired output from the Linker option field. Programs may specify up to 256 user defined section names, and the Linker is capable of processing up to 256 identical section names. See the Linker Description section of this manual for a complete description.

Operating Instructions

Prompt Mode

To run the Assembler type : **x64180**

The Assembler will respond with :

```
Listing Destination ? (N, T, P, D, E, L, <CR=N>):
```

with the abbreviations as follows:

N = None
T = Terminal
P = Printer (Single User Systems Only)
D = Disk
E = Error Only
L = List On/Off

After this the Assembler prompts the operator for the source code filename as shown below.

```
Input Filename :
```

When entering your source filename you may specify an extension or the assembler will look for an extension of '**asm**'. Once you have specified your input filename the assembler will prompt you for the output filename.

```
Output Filename:
```

If the user repsonds to the input filename prompt with just a carriage return, the output file will receive the same filename as the input file, with an extension of '**obj**'. If the repsonse is a filename with no extension, the output file will be under that filename with an extension of '**obj**'.

If the listing is to be under List On/Off Assembler directive control the additional prompt shown below is output:

```
LIST ON/OFF Listing Destination (T, P, D, <CR> = T) :
```

The abbreviations are the same as shown above.

The List On/Off control allows the user to list only selected parts of the source file. For more information see the Listing Control section of this manual.

If Error Only is chosen the Assembler will prompt the user for the destination as follows:

```
Error Only Listing Destination (T, P, D, <CR> = T) :
```

If the listing is being sent to the printer (available on single user systems only) or the disk, the Assembler will prompt for a Cross Reference Listing.

NOTE for VMS users:

Assuming the assembler is located in a directory named $disk1:[x64180], the following command must be entered for the examples shown above to work:

```
x64180 == "$disk1:[x64180]x64180.exe"
```

Command Line Mode

The Assembler may also be invoked using a command line. In this case, the input filename is specified first, then the output filename, and then a list of options. Both the output filename and the listing destination are optional. The general form of the command, with optional fields shown in brackets, is as follows:

```
x64180 [-q] input_filename [output_filename] [-t, -p, -d, -px, -dx]
```

The **-q** optional stands for Quiet mode. If this option is selected, the only screen messages output from the Assembler will be error messages and the line on which they occur. This option must be placed before the input filename.

Below are some examples of legal command lines.

Input Filename Only

```
x64180 input_filename
```

This command causes the Assembler to process the source file '**input_filename**'. If no extension is specified, it is assumed to be '**.asm**'. Since no options are specified, they will default to Error Only listing with the terminal as the destination. The output filename will be the same as the input filename but with an extension of '**.obj**'.

Input Filename and Output Filename

```
x64180 input_filename output_filename
```

This command is identical to the previous one except that the Assembler will name the object file '**output_filename**'.

Listing to Terminal

```
x64180 input_filename output_filename -t
```

This command will assemble the input file '**input_filename**' and send the listing to the terminal. The optional output file name specification causes the assembler to generate an object file named '**output_filename**'.

Listing to Printer

```
x64180 input_filename -p
```

This command will assemble the input file '**input_filename**' and send the listing to the printer. The optional output file name specification causes the assembler to generate an object file named '**input_filename**' with an extenstion of **obj**.

Listing to Printer with Cross Reference

```
x64180 input_filename output_filename -px
```

This command will assemble the input file '**input_filename**' and send the listing and cross reference table to the printer. The optional output filename specification causes the assembler to generate an object file named '**output_filename**'. The printer option is only available on single user systems.

Listing to Disk

```
x64180 input_filename output_filename -d
```

This command will assemble the input file '**input_filename**' and send the listing to the disk. The disk listing file will have the same name as the output file,but will have an extension of '**lst**'.The optional output filename specification causes the assembler to generate an object file named '**output_filename**'.

Listing to Another Drive or Directory

MSDOS

```
x64180  input_filename   output_filename -d,a:
x64180  input_filename   output_filename -d, \new\
```

UNIX

```
x64180  input_filename   output_filename -d, /new/
```

VMS

```
x64180  input_filename   output_filename -d,$disk1:[a]
x64180  input_filename   output_filename -d, [new]
```

This format can be used in any listing mode to send the '**lst**' file to a different directory.

Listing to Disk with Cross Reference

```
x64180 input_filename  output_filename  -dx
```

This command will assemble the input file 'input_filename' and send the listing and cross reference table to the disk. The disk listing file will have the same name as the output file,but will have an extension of 'lst'. The optional output filename specification causes the assembler to generate an object file named '**output_filename**'.

Error Only Listing to the Terminal

```
x64180 input_filename  output_filename  -et
```

This command will assemble the input file '**input_filename**' and send error messages to the terminal. The optional output filename specification causes the assembler to generate an object file named '**output_filename**'.

Error Only Listing to the Printer

```
x64180 input_filename output_filename -ep
```

This command will assemble the input file '**input_filename**' and send error messages to the printer. The optional output filename specification causes the assembler to generate an object file named '**output_filename**'. The printer option is available only on single user systems.

Error Only Listing to the Disk

```
x64180 input_filename output_filename -ed
```

This command will assemble the input file '**input_filename**' and send error messages to the disk. The disk listing file will have the same name as the output file, but will have an extension of '**lst**'. The optional output filename specification causes the assembler to generate an object file named '**output_filename**'.

List On/Off to Terminal

```
x64180 input_filename -lt
```

This command will assemble the input file '**input_filename**' and send **LIST ON/OFF** blocks to the terminal.

List On/Off to Printer

```
x64180 input_filename -lp
```

This command will assemble the input file '**input_filename**' and send the **LIST ON/OFF** blocks to the printer. This option is only available on single user systems.

List On/Off to Disk

```
x64180 input_filename -ld
```

This command will assemble the input file '**input_filename**' and send the **LIST ON/OFF** blocks to the disk. The disk listing file will have the same name as the output file, but with an extension of '**lst**'.

NOTE for VMS users:

Assuming the assembler is located in a directory named $disk1:[x64180], the following command must be entered for the examples shown above to work.

x64180 == "$disk1:[x64180]x64180.exe"

System Defaults

The following default filename extensions will be used by the 2500 A.D. programs if no extension is specified by the user.

Assembler

asm	-	Input to the Assembler
obj	-	Output from the Assembler
pak	-	Packed output from the Assembler
lst	-	Listing file

Linker

obj	-	Input to the Linker
lib	-	Library file
tsk	-	Executable Object Code
hex	-	Intel Hex and Extended Intel Hex
tek	-	Tektronix Hex
s19	-	Motorola S19
s28	-	Motorola S28
s37	-	Motorola S37

Librarian

obj	-	Input to the Librarian
pak	-	Packed input to the Librarian
lib	-	Output from the Librarian

Note that because of the additional information included in the Assembler output file, the Linker must always be run, even if the program is assembled at the desired run address and there are no external references. This is so that all the additional information can be removed and a file with the desired output format can be generated.

Assembler Error Processing

When an assembly error is encountered, the action taken by the Assembler depends on the listing mode it is currently operating under.

If the No List option was specified, the statement causing the error and the error message will be output to the terminal, the display will be turned on and the Assembler will halt just as if the user had typed ^S. The reason for this is to give the user a chance to see exactly where the error is. This will occur on pass 1 as well as pass 2. Note that some errors are not detectable on pass 1, such as undefined symbols. After the error has been displayed, the output can be turned off using ^N.

If the listing is being sent to the printer or the disk, then errors encountered on pass 1 are sent to the terminal but not the printer or disk, and the Assembler does not halt. On pass 2, the error is output to the printer or disk as well as the terminal and the assembly continues.

If the listing is being sent to the printer or disk under assembler directive control, any errors encountered during pass 1 are output to the terminal but not the printer or disk, and the assembly continues. Errors detected during pass 2 are output to the printer or disk and the terminal, even if the error is not inside a block that was specified to be listed.

Assembler Run Time Commands

Assembler Run Time Commands

The following commands are active during the assembly process. These commands are active during pass 1 as well as pass 2, and override the listing mode specified when the Assembler was first activated.

Unix Assembler Run Time Commands

Ctrl S	-	Stop terminal output
Ctrl Q	-	Start terminal output
Del C	-	Terminate the assembly
Del T	-	Display the output at the terminal
Del D	-	Send the output to the disk
Del M	-	Multiple output (Terminal & Disk)
Del N	-	No output

Msdos Assembler Run Time Commands

Ctrl S	-	Stop terminal output
Ctrl Q	-	Start terminal output
Esc C	-	Terminate the assembly
Esc T	-	Display the output at the terminal
Esc P	-	Display the output at the printer
Esc D	-	Send the output to the disk
Esc M	-	Multiple output (Terminal & Disk)
Esc N	-	No output

VMS Assembler Run Time Commands

Ctrl	-	Stop terminal output
Ctrl Q	-	Start terminal output
Ctrl C, C	-	Terminate the assembly
Ctrl C, T	-	Display the output at the terminal
Ctrl C, D	-	Send the output to the disk
Ctrl C, M	-	Multiple output (Terminal & Disk)
Ctrl C, N	-	No output

Assembly Language Syntax

This section describes the syntax used by the 2500 A.D. Cross Assembler.

Number Base Designations

Number bases are specified by the following:

Binary	**-**	**B**
Octal	**-**	**O or Q**
Decimal	**-**	**D or no base designation**
Hex	**-**	**H**
Ascii	**-**	**Single or double quotes - "X" or 'X'**

The two character sequences between single or double quotes shown below are predefined. However, the **TWOCHAR ON** directive must be used to enable these.

"CR"	**or**	**'CR'**	**-**	**Carriage return**
"LF"	**or**	**'LF'**	**-**	**Line feed**
"SP"	**or**	**'SP'**	**-**	**Space**
"HT"	**or**	**'HT'**	**-**	**Horizontal tab**
"NL"	**or**	**'NL'**	**-**	**Null**

Program Comments

Comment lines must start with a semi-colon or asterisk in column 1, unless the **COMMENT** directive is used. Comments after an instruction do not need a semi-colon if at least 1 space or tab precedes the start of the comment if the assembler is running in Spaces Off mode. If the assembler is running in Spaces On mode, all comments after an instruction must be preceded by a semi-colon. See the **SPACES** directive for more information and for the default mode.

Program Counter

The special character dollar sign ($) or asterisk (*) may be used in an expression to specify the program counter. The value assigned to the dollar sign or the asterisk is the program counter value at the start of the instruction.

Labels

Non-Local labels may be any number of characters long, but only 32 characters are significant. Labels may start in any column if the name is terminated by a colon. If no colon is used, the label must start in column 1. All labels must start with an alpha character. Upper and lower case characters are considered to be different.

Local Labels

A Local Label is a label which can be used like any "non local" label. The difference is that the definition of a Local Label is only valid between "non local" labels. The adjective "local" refers to the area between labels which retain their definition through the entire program. When a program passes from one local area to the next, local label names can be reused. This feature is useful for labels referenced only within a "local area", as defined above, and original label names are not necessary.

The assembler identifies a local label by the ($) prefix or suffix. This identifier can be changed with the **LLCHAR** directive. Please see the section entitled 'Directive Definition Control' for more information on this directive. Following are some examples of the use of Local Labels.

```
LABEL1:                    OR      LABEL1:
$1:     NOP                        1$:     NOP
$2:     JMP     $1                 2$:     JMP     1$
        JMP     $2                         JMP     2$

LABEL2:                            LABEL2:
$1:     NOP                        1$      NOP
$2:     JMP     $1                 2$      JMP     1$
        JMP     $2                         JMP     2$

LABEL3:                            LABEL3:
$1:     NOP                        1$      NOP
$2:     JMP     $1                 2$      JMP     1$
        JMP     $2                         JMP     2$
```

In this example, there are three "non-local" labels, LABEL1, LABEL2, and LABEL3. Local Labels, $1 and $2, or 1$ and 2$, have different definitions when referenced in different local areas. Note that $1 is not considered to be the same as 1$. Any character may be used in a Local Label. Local Labels may be up to 32 characters long. Operators such as '+' should never be used in Local Labels. Local labels will not be terminated if the directives **VAR**, **DEFL**, **SECTION**, **ENDS** and **$** are used.

High Byte

To load the high byte of a 16 bit value the unary greater than sign,**>**, should be used. This allows bits 8 through 15 to be used as a byte value which is relocatable.

Low Byte

To load the low byte of a 16 bit value the unary less than sign,**<**, should be used. This allows bits 0 through 7 to be used as a byte value which is relocatable.

Upper / Lower Case

Upper and lower case labels are recognized as different labels. The labels used for section names and macro names are also different if the label is in lower case rather than upper case.

Addressing Modes

Immediate

The data is contained in the instruction.

Examples:

```
LD      HL,1234H    ; Ld HL with the HEX number 1234.
LD      HL,DATA     ; Ld HL with the value associated with the
                      Label 'DATA'.
```

Register

The data is contained in a CPU register.

Examples:

```
LD      A,B         ; Ld the contents in register B into register A
SUB     D           ; Subtract the contents of register D from
                      register A.
```

Register Indirect

The operand address is pointed to by a register.

Examples:

```
LD      A,(HL)      ; Ld A with the contents of the location
                      pointed to by HL.
LD      (HL),B      ; Store the contents of B in the memory
                      address pointed to by HL.
```

Direct Addressing

The address of the operand is contained in the instruction.

Examples:

```
LD      HL,(1234H)      ; Ld HL with the contents of memory location
                          1234 HEX
LD      (ADDRESS),HL    ; Store HL in the memory location 'ADDRESS'
LD      (ABCDH),HL      ; Store HL in the memory location ABCD
                          hexadecimal
```

Indexed

The operand address is the sum of the 8 bit offset in the instruction and the contents of either IX or IY.

Examples:

```
LD      A,(IX+4)        ; Ld A with contents of the memory location
                          pointed to by adding 4 to the contents of
                          register IX
LD      (IX+DATA8),B    ; Store B in the memory location obtained
                          by adding the value associated with 'DATA8'
                          to the contents of register IX
```

Relative

The operand address is relative to the current instruction. If the address is given using a numerical value, the calculation is from the start of the next instruction.

Examples:

```
DJNZ    LOOP            ; The Assembler calculates the address
                          by subtracting the address of the label
                          'LOOP' from the address of the next
                          instruction
JR      4               ; The destination is 4 BYTES past the start
                          of the next instruction
```

Assembler Directives

This section describes the Assembler Directives. Directives may be preceded by a decimal point if desired to help differentiate them from program instructions.

Storage Control

ORG
ORIGIN

Sets the program assembly address. If this directive is not executed, the assembly address defaults to 0000.

END VALUE

This directive defines the end of a program or an included file. The expression following an **END** statement is optional and if it exists, specifies the program starting address. This address is encoded in the output file if a program starting address record type exists in the output format definition.

LABEL: **ASCII** STRING

Stores STRING in memory up to but not including either a carriage return or a broken bar character ("|", Hex 7C). A label is optional. Following are some examples of **ASCII**.

```
ASCII   Hello    ; Stores the Ascii representation of Hello in
                   consecutive memory locations. Incidentally,
                   this comment would be stored also.
ASCII   Hello|   ; Now the comment wouldn't be stored. The next
                   example shows termination with just a carriage
                   return.
ASCII   Hello
```

LABEL: **DB** VALUE
FCB
DEFB
BYTE
STRING

The Assembler will store the value of the expression in consecutive memory locations. The **BYTE** expression may be any mixture of operand types with each one separated by a comma. Ascii character strings must be bracketed by apostrophes. If the string contains an apostrophe, this can be specified with two apostrophes in a row. If no expression is given, one byte is reserved and zeroed. A label is optional. Following are some examples of the use of the **BYTE** directive.

```
.BYTE                ; Reserves 1 zeroed byte.
.BYTE    10          ; Reserves 1 byte = 10 decimal.
.BYTE    1,2,3       ; Reserves 3 bytes,= to 1,2 & 3 in that order.
.BYTE    SYMBOL-10   ; Searches the symbol table for SYMBOL, subtracts
                       10 decimal from it's value, and stores the result.
.BYTE    'Hello'     ; Stores the Ascii equivalent of the string Hello in
                       consecutive memory locations.
.BYTE    'Hello', 0DH ; Same as above example, with the addition of a
                       carriage return at the end. Spaces are ignored before
                       operands, but the comma is required.
.BYTE    '2500 A.D."s' ; Embedded apostrophe.
```

LABEL: **DW** VALUE
FDB
DEFW
LWORD

This directive will store the value of the expression in a 16 bit storage location. Multiple words may be initialized by separating each expression with a comma. If no expression is given, 1 word is reserved and zeroed. A label is optional.

LABEL: **LONG** VALUE
LONGW
LWORD

This directive will store the value of the expression in a 32 bit storage location. Multiple long words may be initialized by separating each expression with a comma. If no expression is given, 1 long word is reserved and zeroed. A label is optional.

LABEL: **FCC** STRING

Stores STRING in memory until a character is reached that matches the first character. The first character and the second matching character are not stored. A label is optional. Typical usage is as follows:

```
FCC     /This is a test string/
```

DC "String"

This directive sets the high bit on the last character of a string.

DS SIZE,VALUE
RMB
DEFS

This directive will reserve the number of bytes specified by SIZE. No value is stored in the reserved area. This directive differs from the BLKB directive in that if the storage locations are at the end of a program section, the output from the Linker is executable, and the Linker is not required to stack another module on top of this section, the reserved bytes are not included in the output file.

LABEL: **FLOAT** VALUE

Converts the value specified into single precision floating point format. The value is not rounded but is truncated if the mantissa is larger than 24 bits. The directive does not allow scientific notation.

```
FLOAT   178.125
FLOAT   100,.125,-178.125
```

LABEL: **DOUBLE** VALUE

Converts the value specified into double precision floating point format. The value is not rounded but is truncated if the mantissa is larger than 52 bits. The directive does not allow scientific notation.

```
DOUBLE  178.125
DOUBLE  100,.125,-178.125
```

LABEL: **BLKB** SIZE, VALUE

Reserves the number of bytes specified by SIZE. If the value field is present, that value is stored in each byte. Otherwise, the reserved bytes are zeroed. A label is optional.

```
BLKB    20          ;Reserves 20 zeroed bytes
BLKB    20,0        ;Reserves 20 zeroed bytes
BLKB    20,FFH      ;Reserves 20 bytes and stores FF Hex in each one
```

LABEL: **BLKW** SIZE, VALUE

Reserves the number of 16 bit words specified by SIZE. If the value field is present, that value is stored in each word. Otherwise, the reserved words are zeroed. A label is optional.

```
BLKW    20          ;Reserves 20 zeroed words
BLKW    20,0        ;Reserves 20 zeroed words
BLKW    20,FFFFH    ;Reserves 20 words and stores FFFF Hex in each one
```

LABEL: **BLKL** SIZE, VALUE

Reserves the number of 32 bit long words specified by SIZE. If the value field is present, that value is stored in each long word. Otherwise, the reserved long words are zeroed. A label is optional.

```
BLKL    20          ;Reserves 20 zeroed long words
BLKL    20,0        ;Reserves 20 zeroed long words
BLKL    20,FFFFH    ;Reserves 20 long words and stores FFFF Hex in each
                    one
```

Definition Control

LABEL: **EQU** VALUE
EQUAL

Equates LABEL to VALUE. VALUE may be another symbol or any legal arithmetic expression.

LABEL: **VAR** VALUE
DEFL

Equates LABEL to VALUE, but may be changed as often as desired throughout the program. A label defined as a variable should not be redefined by an **EQUAL** directive.

LLCHAR CHARACTER

The default character for designating a Local Label is the ($). This directive changes the character which identifies a particular symbol as a Local Label. Symbols that designate number bases should be avoided, unless they are used on the trailing end of the label.

LABEL: **MACRO** ARGS

Specifies the start of a Macro Definition.

ENDM
MACEND

Specifies the end of a Macro Definition.

MACEXIT

This directive causes the immediate exit from a macro. The difference between MACEXIT and MACEND is that during the macro definition process, MACEXIT does not terminate the macro, and if MACEXIT is in the path of a false conditional assembly block, it is not executed. All conditional assembly values are restored to the same state as when the macro was invoked.

MACDELIM CHARACTER

This directive is used to pass an argument containing a comma into a macro. The default mode is for commas to always be argument separators. The allowed characters are '{', '(' and '['. All characters between matching delimiter pairs will be passed through as one argument. Please refer to the Macro Examples section of this manual for some examples of the use of this directive.

XDEF LABEL
GLOBAL
PUBLIC

Specifies the label as a global label that may be referenced by other programs. Multiple labels may be specified as long as each one is separated by a comma. Below are some examples of the correct use of GLOBAL.

```
GLOBAL  SYM1        ; Declares the label SYM1 to be accessible to other programs. The Linker will resolve external references.

GLOBAL  SYM1,SYM2   ; Multiple declarations on the same line are legal separated by a comma. The spaces are ignored.
```

GLOBALS ON

This directive causes the Assembler to treat all labels after GLOBALS ON as global labels which may be referenced by other programs. This directive will not affect Local Labels. Below is an example of the use of GLOBALS.

```
GLOBALS     ON
SYM1        NOP   ;Declares the labels SYM1 and SYM2,
SYM2        NOP   ;accessible to other programs. This directive is not
                  ;reset by the module and endmod directives. The
                  ;default is GLOBAL OFF.
```

GLOBALS OFF

This directive returns the Assembler to the default mode which requires Global symbols to be specified with GLOBAL directives.

XREF LABEL
EXTERN
EXTERNAL

Specifies the label as being defined in another program. Multiple labels may be specified as long as each one is separated by a comma.

LABEL: **ASK** PROMPT

Outputs 'PROMPT' to the terminal and waits for a 1 character response, from which 30 hex is subtracted. The purpose of this is usually to introduce a 0/1 flag into the program. 'LABEL' is set equal to the result. A carriage return terminates 'PROMPT'. On pass 2, the line is output along with the response.

The following is an example of 'ASK' :

```
DISK_SIZE: ASK  ASSEMBLE FOR 8" (=1) OR 5 1/4" (=0) DRIVES ? :
```

Assembly Mode

LABEL: **SECTION**

This directive allows user defined section names to be generated. The Assembler has 2 predefined sections, **CODE** and **DATA**. The total number of section names allowed per file is 256. Each name may be up to 32 characters long. Lower and upper case are considered to be different. After the section has been defined, the program may switch back and forth simply by using the name as a mnemonic. The default section is **CODE**. Sections may be nested. As with all directives, a section name may be preceded by a decimal point. See the Linker Operating Instructions section of this manual for information on how the Linker handles section names. Below are some examples of defining section names and switching between different sections.

```
                NOP         ;This instruction goes into the CODE section
                            ;by default

                .DATA       ;Switch to the predefined DATA section
                .BYTE       ;This byte goes into the DATA section

SECTION1:       .SECTION    ;Define a new section. The definition makes
                            ;this section active automatically

                NOP         ;This instruction goes into the SECTION1
                            ;section

                .CODE       ;Switch back to the section named CODE

                NOP         ;This instruction goes into the CODE section

                .SECTION1   ;Switch to the user defined section SECTION1

                NOP         ;This instruction goes into the SECTION1
                            ;section

                .BYTE       ;Any section may contain code or data or both.
```

ENDS

This directive is used in conjunction with the **SECTION** directive. **ENDS** enables the termination of nested sections in a file.

ABSOLUTE

This directive enables the assembler to use page 0 addresses when possible. This directive is supported for compatibility with our series 3.0 assemblers. For a more detailed discussion, refer to the **"Absolute versus Relative"** section of this manual. Executable instructions should always be assembled in Relative mode.

RELATIVE

This directive enables the assembler to return from Absolute mode to Relative mode. Executable instructions should always be assembled in Relative mode. This is the default mode.

RADIX VALUE

2	**or**	**B**			**= Binary**
8	**or**	**O**	**or**	**Q**	**= Octal**
10	**or**	**D**			**= Decimal**
16	**or**	**H**			**= Hexadecimal**

No expression = return to default mode which is base 10, and assume all others will be designated with B, Q, D or H after the constant. Note that when base 16 is specified there is no way to define a decimal or binary number, since both D and B are legal hexadecimal numbers.

INCLUDE filename

Directs the Assembler to include the named file in the assembly. Filenames may include pathnames. Filename extensions must be completely specified. Includes may not be nested.

SPACES ON

This directive enables spaces in between operands. When spaces are enabled, comments must begin with a semi-colon. The default mode is spaces off.

SPACES OFF

This directive disables spaces in between operands. When spaces are disabled, comments do not need to start with a semi-colon. This is the default mode.

TWOCHAR ON

This directive enables the ascii two character abbreviations shown below. The default mode is **TWOCHAR OFF**.

"CR" or 'CR'	**-**	**Carriage return**
"LF" or 'LF'	**-**	**Line feed**
"SP" or 'SP'	**-**	**Space**
"HT" or 'HT'	**-**	**Horizontal tab**
"NL" or 'NL'	**-**	**Null**

TWOCHAR OFF

This directive disables the ascii two character abbreviations shown in the previous directive. This is the default mode.

MODULE

This directive is meant to be used in conjunction with the **ENDMOD** directive and the Library Manager. Normally, libraries are composed of many small routines. When the Linker cannot find a Global Symbol in any of the files that are involved in the link, it can search the libraries for the symbols it cannot find. This means that each routine must be in a separate file and each file must be assembled separately.

Instead of having separate files, each routine can be bracketed with the **MODULE** and **ENDMOD** directive, which allows all the routines to be in one file. This essentially causes the Assembler to treat each module as a totally separate assembly, so references to External symbols must be declared External, and symbols used by other modules must be declared Global. All modules must be terminated with an **ENDMOD**. Modules may not be nested. Modules may have include files within them, but they may not be inside an include file. There is no limit on the number of modules that may be in a file. The Assembler will produce an output file with an extension of **pak**. This file can only be processed by the Librarian, but is simple to manipulate with the Librarian commands **ADD ALL** and **REPLACE ALL**. Please see the section entitled **Librarian Commands** for information on these commands. Following is an example of the use of **MODULE** and **ENDMOD**.

```
                .MODULE     JUMP_TABLE  ;Define library name
                .GLOBAL     JUMP_TABLE  ;Make table available to other
                                        ;modules in file
                .EXTERN     ROUTINE1    ;Define externals
                .EXTERN     ROUTINE2

JUMP_TABLE:     .WORD       ROUTINE1    ;Store Routine Addresses
                .WORD       ROUTINE2
                .ENDMOD                 ;Define end of module
                .MODULE     ROUTINE1    ;Define library name
                .GLOBAL     ROUTINE1    ;Make routine available

ROUTINE1:       NOP
                .ENDMOD                 ;Define end of module
                .MODULE     ROUTINE2    ;Define library name
                .GLOBAL     ROUTINE2    ;Make routine available

ROUTINE2:       NOP
                .ENDMOD                 ;Define end of module
                .END                    ;Define end of file
```

If the above file was named **test.asm**, it would be assembled as usual but the output filename would be **test.pak**. Note during the assembly how the Assembler **restarts** at the beginning of each module.

ENDMOD

This directive is used in conjunction with the **MODULE** directive and terminates each module in a file. Please refer the **MODULE** directive for examples of the use of **ENDMOD**.

COMMENT CHARACTER

This directive allows the user to write blocks of comments at time. A comment block is executed as follows:

```
COMMENT     X
```

Where **X** can be any character. The Assembler will treat everything from the first **X** to the second **X** as a comment block. Since the terminating character is not scanned for until the next line the comment field must be two lines long.

BIT7 ON

This directive will causes the Assembler to set the high bit of each character in an Ascii String. This applies to the **ASCII** directive and the **BYTE** directive only, and it only applies to the BYTE directive when the characters are enclosed in single or double quotes. In otherwords, data values will not be affetected. The Assembler defaults to **BIT7 OFF**.

BIT7 OFF

This directive returns the Assembler to it's default mode, which is to leave bit 7 cleared on Ascii characters.

Conditional Assembly

IFZ VALUE
IFE

The Assembler will assemble the statements following the directive up to an ELSE or ENDIF directive if the VALUE is equal to zero. Conditional statements may be nested up to 248 levels. VALUE can be an arithmetic expression, another symbol or a string.

IF VALUE
IFN
IFNZ
COND

Assemble the statements following the directive up to an ELSE or ENDIF directive if the value of VALUE is not equal to zero. Conditional statements may be nested up to 248 levels.

IFTRUE VALUE
IFNFALSE

This directive is actually the same as IFNZ, but is more logical when using assembly time comparisons. If the specified condition is true, then the following statements are assembled up to an ELSE or ENDIF directive. If the condition is not true, the statements up to an ELSE or ENDIF directive are not assembled.

IFNTRUE VALUE
IFFALSE

This directive is the same as IFZ, and is the complement to IFTRUE. If the specified condition is false, then the following statements are assembled up to an ELSE or ENDIF directive. If the condition is true, then the statements up to an ELSE or ENDIF directive are not assembled.

IFDEF LABEL

This directive will activate a symbol table search, and if LABEL is found, then the statements following this one up to an ELSE or ENDIF directive will be assembled. If LABEL is not found, then the statements following this statement up to an ELSE or ENDIF directive will not be assembled.

IFNDEF LABEL

This directive is the complement of IFDEF. The symbol table is searched and if LABEL is not found, the statements following this one up to an ELSE or ENDIF directive are assembled. If LABEL is found, then the statements following this one up to an ELSE or ENDIF directive are not assembled.

IFSAME STRING1,STRING2
IFNDIFF

This directive compares STRING1 to STRING2, and conditionally assembles the statements following this statement depending on the result of the comparison. If the two strings are identical then the statements up to an ELSE or ENDIF directive are assembled. If the strings are not identical, then the statements up to an ELSE or ENDIF directive are not assembled. The strings may be one of two different types, namely with spaces or without spaces. However, both strings being compared must be of the same type. If the strings contain spaces, then the beginning and end of each string must be denoted with an apostrophe, with embedded apostrophes denoted by the use of two apostrophes. If the strings do not contain spaces, then the apostrophes are not required. This mode is very useful when comparing macro parameter arguments. In both cases, the strings must be separated with a comma. Following are some examples of the use of **IFSAME**.

```
IFSAME      'test string','test string'
IFSAME      '2500 A.D."s','2500 A.D."s'
IFSAME      X,Y
```

In the first example above, the strings contain spaces and therefore must be bracketed by apostrophes. The second example shows embedded apostrophes, which are represented by using two apostrophes. In the third example, a macro might be testing for a certain register, and since the strings do not contain spaces, they do not need to be enclosed in apostrophes.

IFNSAME STRING1,STRING2
IFDIFF

This directive is the complement to IFSAME. If the two strings are not identical, the statements after this statement are assembled up to an ELSE or ENDIF directive. If the two strings are identical the statements up to an ELSE or ENDIF directive are not assembled. The syntax rules governing the form of the strings are the same as for IFSAME. See IFSAME for examples of the use of this directive.

IFEXT LABEL

This directive will cause the Assembler to search the symbol table for the label, and assemble the statements following this statement up to an ELSE or ENDIF if the label has been declared external. An error message is generated if the label is not found.

IFNEXT LABEL

This directive will cause the Assembler to search the symbol table for the label, and assemble the statements following this statement up to an ELSE or ENDIF if the label has not been declared external. An error message is generated if the label is not found.

IFABS LABEL
IFNREL

This directive will cause the Assembler to search the symbol table for the label, and assemble the statements following this statement up to an ELSE or ENDIF if the label is absolute (i.e. not relocatable). External labels are considered to be relocatable. An error message is generated if the label is not found.

IFREL LABEL
IFNABS

This directive will cause the Assembler to search the symbol table for the label, and assemble the statements following this statement up to an ELSE or ENDIF if the label is relocatable. External labels are considered to be relocatable. An error message is output if the label is not found.

IFMA EXP

This directive is intended to be used inside a macro, and will scan the macro call line for the existence of the argument number specified by the value of EXP. If the argument exists, the statements following this one up to an ELSE or ENDIF will be assembled. If the argument does not exist, the statements fol lowing this one up to an ELSE or ENDIF will not be assembled. No arguments can be detected by having EXP = 0. In this case, if no arguments are present in the macro call line, the following statements are assembled, and if arguments are present in the macro call line, the following statements are not assembled. See the Macro section of this manual for examples of the use of this directive.

IFNMA EXP

This directive is the complement to IFMA, and checks the macro call line to see if the argument number given by the value of EXP exists. If the argument is not present, the statements following this one up to an ELSE or ENDIF are assembled. If the argument is present, the statements following this up to an ELSE or ENDIF are not assembled. The existence of any arguments at all can be detected by

having EXP = 0. In this case, if there is at least one argument in the macro call line, the following statements will be assembled. If there are no arguments in the macro call line, the following statements will not be assembled. See the Macro section of this manual for examples of the use of this directive.

ELSE

Start of statements to be assembled if any of the above IF type of directives are false.

ENDC
ENDIF

Specifies the end of a conditional assembly block. When the Assembler detects unmatched IF - ENDIF pairs, an error message is output. Since recursive macros will almost always be controlled by IF type directives, the IFCLEAR directive may be needed. The difference between the two is that ENDIF is always executed, while IFCLEAR is not executed when it is inside a false conditional assembly block.

IFCLEAR

This directive performs exactly the same function as ENDIF, except that it is not executed when it is inside a false conditional assembly block. This directive can be used in a recursive macro to maintain balanced IF - ENDIF pairs, allowing the macro to eventually terminate, yet still taking advantage of the IF - ENDIF checking performed by the Assembler. This directive can be used to perform the same function when a macro contains a MACEXIT directive for early macro exits, since these would almost always be controlled by an IF directive of some sort. See the Macro section of this manual for examples of the use of this directive.

EXIT "MESSAGE"

This directive is meant to be used inside of a conditional and will terminate the assembly if it is executed. **MESSAGE** is output by the assembler as an error message. If the surrounding condition is true, then the **EXIT** directive is executed, the user defined error message is output, and the assembly is terminated. If the surrounding conditional is false, then the assembly coutinues without interruption. The maximum length of the user defined error message is 79 characters. An example of **EXIT** is as follows:

```
IFTRUE    TABLE_SIZE.UGT.MAX_TABLE_SIZE
EXIT
ENDIF
```

NOTE: If the assembly is terminated, it will occur on the first pass and no listing file will be created.

Assembly Listing Control

LIST ON
LIST

Turns listing on if **LIST ON/OFF** was specified as the listing destination when the Assembler was first entered. This directive must always be used before **LIST OFF**. In other words, at the start of the program, **LIST OFF** is assumed.

LIST OFF
NOLIST
NLIST

Turns listing off if **LIST ON/OFF** was specified and **LIST ON** was executed. This is the default mode and therefore should only be used following a **LIST ON** directive.

MACLIST ON
MLIST

Turns listing of MACRO expansions on. This is the default mode.

MACLIST OFF
MNLIST

Turns listing of MACRO expansions off. The default is on.

CONDLIST ON

Turns on listing of false conditional assembly blocks. This is the default mode.

CONDLIST OFF

Turns off listing of false conditional assembly blocks. The default is on.

ASCLIST ON

Turns on the listing of ascii strings that require more than 1 line of object code on the assembler listing.

ASCLIST OFF

Turns off the listing of ascii strings that require more than 1 line of object code on the assembler listing. Only the first line of the object code will be listed.

PW EXP

Sets the printer page width. The default page width is 132 columns.

PL EXP

Sets the printer page length. The default page length is 61 lines. The Assembler issues a form feed when this limit is reached or exceeded. If an error is encountered, the Assembler will output the form feed after the error message.

TOP EXP

This directive controls the number of lines from the top of the page to the page number. The default is zero.

PASS1 ON

Turns on the listing of pass 1. This can be used to help find errors due to the Assembler taking a different path on Pass 1 as compared to Pass 2. This condition will usually generate a 'Symbol value changed between passes' error. This directive can also be useful for finding nested conditional assembly errors.

PASS1 OFF

Turns off listing of pass 1 assuming **PASS1 ON** was executed.

PAG
PAGE
EJECT

Outputs a form feed to the listing device.

NAM STRING
TTL
TITLE
HEADING

Causes **STRING** to be printed at the top of every page. If **STRING** is not specified the **TITLE** directive will be turned off. The title may be changed as often as desired and may be turned off at any time. The maximum title length is 80 characters. Also, the first two tabs between the **TITLE** directive and the start of the string, if they exist, will be ignored. All spaces and tabs after this will be included in the title.

STTL STRING
SUBTITLE
SUBHL

Causes **STRING** to be printed at the top of every page. If **TITLE** was executed, the subtitle will appear below it. If **TITLE** was not executed or was turned off, the subtitle will still be output. If **STRING** is not specified, the directive will be turned off. The subtitle may be changed as often as desired and may be turned off at any time. The maximum subtitle length is 80 characters. As with the **TITLE** directive, the first two tabs between the **SUBTITLE** directive and the start of **STRING**, if they exist, will be ignored and any spaces and tabs that appear after that will be included in the subtitle.

Linker Control

FILLCHAR VALUE

The linker will fill in gaps which are created by the use of sections or origins with the value specified. This directive is only applicable to the executable output from the Linker. All other output formats will begin a new recored if an origin gap is detected.

RECSIZE VALUE

The record length may be changed for Intel Hex and Motorola S record outputs with this directive. By specifying a value standard 32 data bytes for Intel and 131 data bytes for Motorola will be replaced with VALUE.

SYMBOLS

This enables the symbols to be sent to an output file for the linker. This directive must be used to enable the Linker to output the Microtek symbol table format.

OPTIONS OPTION LIST

This directive is used to select the options for the Linker. For a list of the options see the Linker Options section of this manual. The default output filetype is Intel Hex. The output from the Linker may still be changed by using the Linker options field.

LINKLIST

This directive will cause the linker to relocate the assembler listings so that the execution address, the addresses in the object code field and the values in the cross reference table are the actual vlaues at run-time. This directive works with the listing to disk option only.

COMREC "String"

This directive allows the user to insert a comment record in the Motorola outputs. The format of **COMREC** is as follows.

COMREC "STRING"

Assembly Time Calculations

The following list gives the allowed assembly time calculations. Also shown is their priority level. Priority level 7 operations are the first to be performed. Parenthesis may be used to force the calculations to proceed in a different order. Calculations are performed using 80 bit integer arithmetic with the exception of exponentiation which only uses an 8 bit exponent. The maximum number of pending operations is 16.

OPERATION	PRIORITY	DESCRIPTION
Unary +	7	Optionally specifies a positive operand.
Unary -	7	Negates the following expression.
\ or .NOT.	7	Complements the following expression.
Unary >	7	Keeps the high order byte of the following address. This must be used to obtain relocatable byte address values.
Unary<	7	Keeps the low order byte of the following address. This must be used to obtain relocatable byte address values.
**	6	Unsigned exponentiation
*	5	Unsigned multiplication
/	5	Unsigned division
.MOD.	5	Remainder
.SHR.	5	Shift the preceding expression right (with 0 fill) the number of times specified in the following expression.
.SHL.	5	Shift the preceding expression left (with 0 fill) the number of times specified in the following expression.
+	4	Addition
-	4	Subtraction
& or .AND.	3	Logical AND
^ or .OR.	2	Logical OR
.XOR.	2	Logical exclusive OR

Assembly Time Comparisons

The following list gives the assembly time comparisons which will return all 1's if the comparison is true and all 0's if the comparison is false:

=	or	.EQ.	-	Equal
>	or	.GT.	-	Greater than
<	or	.LT.	-	Less than
		.UGT.	-	Unsigned greater than
		.ULT.	-	Unsigned less than

Absolute Versus Relative

The absolute directive enables the assembler to use page 0 addresses when possible and should be used when a symbol is required to have an absolute value. This directive is supported for compatibility with our series 3.0 assemblers. If the Absolute directive is used, the Relative directive must be used to return the assembler to relocatable mode. The assembler should always be returned to Relative mode before any executable instructions are assembled. The Absolute & Relative attributes do not change when the section is changed.

Another valid use of this directive is in laying out assembly language structures. This can be done in a user defined section as in the following example:

```
STRUCTURE_SECTION:  .SECTION
                    .ABSOLUTE
                    .ORIGIN     <initial offset>
NAME:               .DS         <expression>
COMPANY:            .DS         <expression>
ADDRESS:            .DS         <expression>
CITY:               .DS         <expression>
STATE:              .DS         <expression>
ZIP_CODE:           .DS         <expression>
STRUCTURE_SIZE:     .DS         0
```

where the Origin statement may be omitted if the "initial offset" for the structure is zero and "expression" is equal to the size of the corresponding member of the structure. Note that in this example, storage space for three different structures of this type could be reserved by the following code:

```
STRA:   .DS     STRUCTURE_SIZE
STRB:   .DS     STRUCTURE_SIZE
STRC:   .DS     STRUCUTRE_SIZE
```

Forming structures in this way has the advantage of automatically computing offsets and structure sizes while allowing the programmer to add or delete elements of the structure without re-computing the offset for each individual member. If this section is linked as a "reference only section" by preceding the "load offset" given at link time with a hyphen, then the bytes reserved by the DS directives will not be included in the output file. For more information regarding "reference only" refer to the linker section of this manual.

Note that the correct procedure for generating executable code at absolute addresses is:

(1) Assemble in relative mode.

(2) Supply the linker with a "load offset" of zero for these instructions at link time.

If executable instructions are assembled in Absolute mode, relative references will be calculated with absolute values. The result of this is that displacements will be an absolute number, just as if the symbol was defined with the **EQUAL** directive. Consider the following example:

```
                .CODE
                .ABSOLUTE
                .ORG        20H
LABEL:          NOP
                BRA         LABEL
                .END
```

where LABEL has a value of 20H. This **BRA** instruction will use a +20H as its displacement value. This is equivalent to the instruction:

```
                BRA         20H
```

where the next instruction executed will always be +20H bytes away from this **BRA** instruction.

Macros

Definition

A macro is a sequence of source lines that will be substituted for a single source line. A macro must be defined before it is used. The Assembler will store the macro definition and, upon encountering the macro name, will substitute the previously defined source lines. Arguments may be included in the macro definition. Arguments may be substituted into any field except the comment field.

For macro definitions, dummy arguments may not contain spaces. However, for actual macro calls, arguments may be any type; direct, indirect, character string or register. Spaces are not allowed in arguments unless it is an Ascii string, in which case the string must be bracketed in apostrophes. If the string contains an apostrophe, this can be specified with two apostrophes in a row. Arguments will be passed through to any nested macros if the dummy argument names are identical. Macro nesting is limited only by the amount of memory space available.

To define a macro the **.MACRO** directive is used. A macro must have the **.MACEND** or **.ENDM** directive following the macro definition. The name of the macro is in the label field.

Argument Separators

In the macro call line arguments must be separated by commas, however leading spaces and tabs are ignored. If no argument is present, a single comma will serve as a place holder.

The * as an argument will not be used as the program counter but as the multiplication sign. In a macro body, the following argument separators are allowed:

```
, + - * / ** \ & ^ = ( ) [ ] |
.NOT. .AND. .OR. .XOR. .EQ. .GT. .LT .GT.
.ULT. .SHR. .SHL.
```

Labels In Macros

Labels are allowed in macro definitions. Labels may be defined in two ways: explicit or implicit. Explicit labels in the macro definition will not be altered by the Assembler. Implicit labels are followed by a #. The Assembler will substitute a 3 digit macro expansion number for the #. In this case, the label and the macro expansion number must not exceed 32 characters. An argument may be used to specify a label.

String Concatenation

The broken bar character (| = hex 7C) is used as the string concatenation operator. Concatenation may only be performed inside of a macro.

Value Concatenation

Concatenation of a string and the value of an expression may be achieved by using the broken bar character (| = hex 7c) followed by a left angled bracket, the expression, and a right angled bracket. No spaces are allowed between the broken bar and the left angled bracket. Following is an example of this operation:

```
CONCAT          .MACRO      ARG
VALUE:          .VAR        VALUE+1
ARG|<VALUE*2>   .EQU        31
                .ENDM
VALUE           .VAR        0
                CONCAT      LABEL
```

The invocation, CONCAT LABEL, will produce:

```
LABEL2          .EQU        31
```

It is important to initialize VALUE before the macro is invoked. Otherwise, the label being generated will have a different value on pass 1 and pass 2.

Mnemonic Definitions

The Assembler tables are searched in the following order:

1st	**-**	**Mnemonic Table**
2nd	**-**	**Macro Definition Table**
3rd	**-**	**Assembler Directive Table**
4th	**-**	**Section Name Table**

To redefine a mnemonic the MACFIRST directive may be used. This will switch the order of the search to Macro Definition Table first and Mnemonic Table second.

Macro Examples

A macro could be written to do string comparisons. This macro demonstrates the use of this feature.

```
CMP_STRING:     .MACRO      ARG1
                IFNMA       1
                CMP_STRING NEEDS AN ARGUMENT
                MACEXIT
                ENDIF
                IFSAME      "JANUARY",ARG1
MONTH           BYTE        1
                MACEXIT
                ENDIF
                IFSAME      "FEBRUARY",ARG1
MONTH           BYTE        2
                MACEXIT
                ENDIF
                IFSAME      "MARCH",ARG1
MONTH           BYTE        3
                MACEXIT
                ENDIF
                IFSAME      "APRIL",ARG1
MONTH           BYTE        4
                MACEXIT
                ENDIF
                IFSAME      "MAY",ARG1
MONTH           BYTE        5
                MACEXIT
                ENDIF
                IFSAME      "JUNE",ARG1
MONTH           BYTE        6
                MACEXIT
                ENDIF
                ARGUMENT ERROR IN MACRO STRING
                ENDM
                CMP_STRING  "APRIL"
                END
```

The following example demonstrates the use of argument substitution in the operand field of a macro.

```
EMPLOYEE_INFO:    .MACRO              ARG1,ARG2,ARG3
NAME:             .DB                 ARG1
DEPARTMENT:       ASCII               ARG2
DATE_HIRED:       .LONG               ARG3
                  .ENDM
                  EMPLOYEE_INFO       'JOHNDOE',PERSONNEL,101085
                  .END
```

This example could be changed to pass the argument into the label field. This enables the structure to be altered.

```
EMPLOYEE_INFO:    .MACRO              ARG1,ARG2,ARG3
ARG1:             .DS                 30H
ARG2:             .DS                 10H
ARG3:             .LONG
                  .ENDM
                  EMPLOYEE_INFO NAME,DEPARTMENT,DATE_HIRED
                  .END
```

The macro section also allows substitution into the mnemonic field. Also, a label can be generated within the macro with the # sign.

```
INSTRUCTION:      MACRO               ARG,VAL
                  ARG
LAB#:             DS                  VAL
                  .MACEND
                  INSTRUCTION NOP,7
                  .END
```

To redefine a mnemonic the **MACFIRST ON** directive must precede the macro.

```
                MACFIRST    ON
NOP:            .MACRO      ARG
                DB          ARG
                .ENDM
                NOP         FFH
                END
```

Another macro directive, **MACDELIM**, can be used to pass commas into a macro. The following examples show the syntax for this directive.

```
                MACDELIM    {
DELIM_EX:       MACRO       ARG1  ARG2
                BYTE        FFH,ARG1,ARG2
                ENDM
                DELIM_EX    {,A4H},{,12H}

                MACDELIM    [
DELIM_EX:       MACRO       ARG1
                BYTE        FFH    ARG1
                ENDM
                DELIM_EX    [,A4H]

                MACDELIM    (
DELIM_EX:       MACRO       ARG1
                BYTE        FFH    ARG1
                ENDM
                DELIM_EX    (,A4H)
```

Recursion

Below is an example of a recursive macro that reserves the number of data bytes defined by dummy argument ARG1 and fills them with the value specified by ARG2,ARG3,ARG4,ARG5,ARG6. This also demonstrates the use of **MACEXIT** and **IFCLEAR**. Count is decremented each time the loop is executed successfully. The macro is called again with the statement RESERVE the arguments following.

```
RESERVE: .MACRO     ARG1,ARG2,ARG3,ARG4,ARG5,ARG6
COUNT:   .VAR       ARG1
         .IFZ       COUNT
         .IFCLEAR
         .MACEXIT
         .ENDIF
COUNT:   .VAR       COUNT-1
         .BYTE      ARG2,ARG3,ARG4,ARG5,ARG6
         RESERVE    COUNT,ARG2,ARG3,ARG4,ARG5,ARG6
         .MACEND
```

This macro would be called with a statement such as the following:

```
RESERVE 10,AH,BH,CH,DH,EH          ; Fill 50 bytes with the sequence ABCDE.
```

It is perfectly legal for a recursive macro, such as the one in the above example, to call another recursive macro and so forth out to whatever level is desired. Also, note the use of the **IFCLEAR** directive, which maintains the conditional **IF - ENDIF** pair balance. This can be used but is not required because the **MACEXIT** directive will return all conditionals to their original state.

Assembler Error Messages

Error	- CAN'T CREATE OUTPUT FILE - DISK MAY BE FULL
Meaning	-The disk may actually be full or the operating system is not allowing enough files to be open at one time. See System Requirments to correct this error.

Error	- CAN'T OPEN INPUT FILE
Meaning	- The operating system is not allowing enough files to be open at one time. See System Requirements to correct this error.

Error	- CAN'T FIND FILENAME.OBJ
Meaning	- The .OBJ filename does not exist or the operating system is not allowing enough files to be open at one time. See System Requirements to correct this error.

Error	- SYNTAX ERROR
Meaning	- Usually a missing comma or parenthesis.

Error	- CAN'T RESOLVE OPERAND
Meaning	- Can't tell what the programmer intended.

Error	- ILLEGAL ADDRESSING MODE
Meaning	- Can't address the operand using this form.

Error	- ILLEGAL ARGUMENT
Meaning	- Operand can't be used here.

Error	- MULTIPLY DEFINED SYMBOL
Meaning	- Symbol defined previously (not including '.VAR')

Error	- ILLEGAL MNEMONIC
Meaning	- Mnemonic doesn't exist and wasn't defined as a Macro.

Error	- # TOO LARGE
Meaning	- The destination is too small for the operand.

Error	- ILLEGAL ASCII DESIGNATOR
Meaning	- Bad punctuation on Ascii character.

Error	- HEX # AND SYMBOL ARE IDENTICAL
Meaning	- A label exists that is exactly identical to a hex number that is being used as an operand. Even the hex number indicator must be in the same place for this error to be generated.

Error	- UNDEFINED SYMBOL
Meaning	- Symbol wasn't defined during pass 1.

Error	- RELATIVE JUMP TOO LARGE
Meaning	- Destination address in a different page.

Error	- EXTRA CHARACTERS AT END OF OPERAND
Meaning	- Usually a syntax or format error.
Note	- This error is the last check on any instruction before the Assembler proceeds to the next line and indicates that there are extra characters after a legal operand terminator.

Error	- LABEL VALUE CHANGED BETWEEN PASSES
Meaning	- Symbol value decode during pass 1 not = pass 2.
Note	- This error is usually caused by the Assembler taking different paths on Pass 1 as compared to Pass 2 due to conditional directive arguments changing value. The directive PASS1 ON/OFF can be useful in finding these types of errors.

Error	- ATTEMPTED DIVISION BY ZERO
Meaning	- Divisor operand evaluated to 0.

Error	-ILLEGAL EXTERNAL REFERENCE
Meaning	-External reference can't be used here.

Error	-NESTED CONDITIONAL ASSEMBLY UNBALANCE DETECTED
Meaning	-Any '.IF' type instruction without a matching '.ENDIF'

Error	- ILLEGAL REGISTER
Meaning	- The specfied register is not legal for the instruction

Error	- CANT RECOGNIZE NUMBER BASE
Meaning	- The number base specified is not one the assembler accepts.

Error	- NOT ENOUGH PARAMETERS
Meaning	- The were more arguments than parameters in a macro.

Error	- ILLEGAL LABEL 1ST CHARACTER
Meaning	- Labels must start with an alpha character.

Error	- MAXIMUM EXTERNAL SYMBOL COUNT EXCEEDED
Meaning	- There were too many externals in a module.
Note	- There is a maximum of approximately 500 externals per module.

Error	- MUST BE IN SAME SECTION
Meaning	- The instructions operand is in a different section.

Error	- NON-EXISTENT INCLUDE FILE
Meaning	- The include file could not be found.

Error	- ILLEGAL NESTED INCLUDE
Meaning	- One included file contains an .INCLUDE directive. This error may also indicate that an included file did not have an END statement.

Error	- NESTED SECTION UNBALANCE
Meaning	- A nested section definition without an ENDS

Error	- MISSING DELIMETER ON MACRO CALL LINE
Meaning	- Unmatched delimeters when a macro was invoked.

Error	- MULTIPLE EXTERNAL IN THE SAME OPERAND
Meaning	- More than one external exists in the same operand.

Error	- A LABEL IS ILLEGAL ON THIS INSTRUCTION.
Meaning	-This is used to flag labels that would not obtain a relocation value. Such as ENDM or MACEND. Thus, the label is not allowed for the instruction.

Error	- MACRO STACK OVERFLOW
Meaning	- Macros are nested too deeply.
Note	- This error can be caused by too many recursive macro calls. The stack has room for approximately 700 nested or recursive macro calls.The number of calls is affected by the number of arguments the macro uses.

Error	- MISSING LABEL
Meaning	- A label is required for this instruction.

Error	- OPERAND MUST BE DEFINED AS AN 8 BIT RELOCATABLE VALUE.
Meaning	- This occurs when a 16 bit address is used in an 8 bit instruction. The < or > sign must be used to make the value relocatable.

Error	- MISSING RIGHT ANGEL BRACKET
Meaning	- Right angle bracket is mandatory.

Error	- MACRO NAME MUST APPEAR ON SAME LINE AS MACRO DEFINITION

Error	- ILLEGAL LOCAL LABELS
Meaning	- Labels can't be defined as local. For example .VAR.

Error	- MISSING MODULE DIRECTIVE

Error	- MISSING ENDMOD DIRECTIVE

Error	- 'Module' CAN'T BE IN 'Include' FILE

Error	- 'Endmod' CAN'T BE IN 'Include' FILE

LINKER

2500 A.D. Linker Description

The 2500 A.D. Linker enables the user to write assembly language programs consisting of several modules. The Linker will resolve external references and perform address relocation. The Linker is capable of generating all of the most used file formats, eliminating the need for an additional format conversion utility.

Except for when generating an executable output file, the Linker runs entirely in RAM. There is no limit on the size of file that can be linked, as long as enough memory is available. In the case of an executable file, the Linker creates as many scratchpad files as required to sort the different program sections in ascending order.

Each object file may have up to 256 different user defined sections. The Linker can search up to fifty separate library files for resolving external symbol references. The Linker can process a combination of 256 input files and library modules, and 256 different section names. There is no limit on the size of each section.

Files may be linked at the address specified in the file or relocated at link time. Specific sections of files may be used for reference only. That is, the information from the section needed to link will be used but the section will not be included in the output file. Sections of files may also be linked at different run-time and load addresses. This feature can be used to generate romable code that must be moved to read/write memory at run time. For more information see the section entitled **Indirect Linking.**

Listings with the Linker may be relocated using **LINKLIST**. This directive, when listing to disk is specified, will relocate the listing with the actual addresses at Run-Time. Please see the **Linker Control** section for more information.

The Linker may be invoked using Prompt mode, Command Line mode or Data File mode. The output format is selectable from a directive in the source file or from the Linker options list. The Load Map, an alphabetized global symbol list and all link errors may be saved in a disk file.

The Linker may be directed to output several different types of symbol table files. These formats are relocated 10 character global symbols, relocated 32 character global symbols, and the Microtek format which includes all symbols.

An environment variable may be defined to specify a search path for library files. The environment variable name is **LIB**. If the variable **LIB** is already defined, the path names represented by the variable should be redefined. Please refer to the operating system manuals for information on defining the **LIB** environment variable.

Linker Operating Instructions

Prompt Mode

To run the Linker in prompt mode type **Link**. The Linker will respond with a prompt requesting an input filename. The default extension for a Linker input file is 'obj'. After opening the object file, the Linker will prompt for the offset address for each program section that has a non-zero size. This offset value is added to the value of any **ORG** statements in the file. A carriage return only response will cause the Linker to stack each program section on top of the preceding section. A minus sign causes the Linker to relocate the section, but not include it in the output file. A semi-colon after any offset address causes the Linker to automatically stack each section on top of the previous section. Since the best way to explain all of this is with examples, please refer to the Linker Examples section of this manual.

The input phase can be terminated by responding to the input filename prompt with just a carriage return. The Linker will then prompt for an output filename. A carriage return only response to the output filename prompt will cause the Linker to generate an output file with the same name as the first input file and an extension that is determined by the output file type.

After the output filename has been entered, the Linker will prompt for library filenames. The Linker can search up to **50** libraries for external symbol references. A carriage return only response to the library filename prompt will terminate library filename input.

After any library filenames have been entered, the Linker will prompt for any Linker options. The Linker options are described in the section entitled Linker Options.

NOTE to VMS users:

Assuming the linker is located in a directory named $disk1:[link], the following command must be entered for the examples shown above to work:

```
link == "$disk1:[link]link.exe"
```

if you use the VMS Link program, one of the linkers should be renamed.

Data File Mode

Data File mode is included for large or complex linking. This mode can be viewed as being identical to prompt mode, except that all of the responses to the prompts are placed in a file and the file is submitted to the Linker. The command is as follows:

```
Link data_file
```

This causes the Linker to read the file **data_file.lnk** and uses the responses in the file, line by line. The Linker assumes an extension of **lnk** on the data file. Since carriage return only responses may be difficult to see in a data file, an underbar character ('_') may be placed on a carriage return only line. If Linker options are specified, they are placed last in the file, just as they are in prompt mode. The following sample Data File will link 2 files together with the section named **CODE** starting at 2000H and the section named **DATA** starting at 4000H. The default Linker output filename is to be used, and the D and 3 options are used to generate a disk map file and a Motorola S37 output file.

```
file1       First input filename
2000        Put the CODE section at 2000H
4000        Put the DATA section at 4000H
file2       Second input filename
_           Stack CODE section on top of 1st CODE
_           Stack DATA section on top of 1st DATA
_           No more input filenames
_           Use default output filename
_           No library filenames
d3          Create disk file & Motorola S37 file
```

The easiest way to construct a Data File is to run through the link process in Prompt mode, and write down each response. Then, using a text editor, create a file with each response on a line by itself. This file should have an extension of **lnk.**

Any line that has a semi-colon or an asterisk in column 1 will be considered to be a comment line.

NOTE to VMS users:

Assuming the linker is located in a directory named $disk1:[link], the following command must be entered for the examples shown above to work:

```
link == "$disk1:[link]link.exe"
```

If you use the VMS Link program, one of the linkers should be renamed.

Command Line Mode

The Linker may be invoked by using a command line. The form of this command is shown below, with optional fields shown in brackets.

```
Link [-q] -c file1 [-Innnn] file2 [-Innnn] ...[-ofile] [-options]
```

The **-q** option puts the Linker in Quiet mode. In this case the only output to the terminal from the Linker are link errors.

The **-c** option is required, and informs the Linker that it is running in Command Line mode instead of Data File mode.

Following the **-c** is the list of input files, denoted in the above command line by **file1** and **file2.** Each input file may be followed by an offset address by using the **-I** option. If the address offset is not included, each file is stacked on top of the previous file according to matching section names.

The **-o** option can be used to specify an output filename. This field is optional. If no output filename is specified, the Linker will create an output file with the same name as the first input file, with an extension determined by the output file format.

The **-L** option can be used to specify library filenames. A maximum of **50** library filenames can be specified.

The **options** field allows any of the Linker options to be specified. A minus sign is required in front of the list, and as many options as desired may be specified. See the Linker Options section of this manual for a description of the options.

NOTE to VMS users:

Assuming the linker is located in a directory named $disk1:[link], the following command must be entered for the examples shown above to work:

```
link == "$disk1:[link]link.exe"
```

If you use the VMS Link program, one of the linkers should be renamed.

Linker Options

In prompt mode, the linker options prompt appears after the output filename prompt. The options below are also available in Command Line and Data File mode. When more than one option is specified the final option will override the previous options.

Options (D, S, A, M, Z, X, H, E, T, 1, 2, 3, <CR> = Default)

D - Create a disk file containing any link errors, an alphabetized global symbol table, and the Load Map. The file created has the same name as the Linker output file with an extension of '**map**'.

S - Create a symbol file for debugging purposes. The file contains all the global symbols and relocated values. Each symbol is 32 characters in length. See the Symbol Table Output Format section of this manual for exact details.

A - Create a symbol file for debugging purposes, but limit the symbols to the first 10 characters. This is used for compatability with the 3.0 2500 A.D. series of Linkers. See the Symbol Table Output Format section of this manual for exact details.

M - Create a symbol file for debugging purposes in the Microtek format. This file includes all symbols, both local and global. The **SYMBOLS ON** directive must be included in the source file for this format to be generated.

Z - Create a symbol file for debugging purposes in the Zax format.This file includes all symbols, both local and global. The **SYMBOLS ON** directive must be included in the source file for this format to be generated.

X - Generate an Executable output file.
H - Generate an Intel Hex output file.
E - Generate an Extended Intel Hex output file.
T - Generate a Tektronix Hex output file.
1 - Generate a Motorola S19 output file.
2 - Generate a Motorola S28 output file.
3 - Generate a Motorola S37 output file.

Address Relocation

Addresses are relocated by adding the offset address to the address decoded by the 2500 A.D. Assembler. Normally the program would be assembled starting at location 0000, but it doesn't have to be. The offset address will simply be added to any address generated by the Assembler.

The Assembler maintains a table of attributes associated with each symbol used in the program. If the label simply precedes an instruction, then it is tagged as relocatable. If the label is defined in an **.EQUAL** directive, then the relocatability of it depends on the operand field type. If the operand contains no relocatable tokens, then the expression is not relocatable. If the operand contains only one relocatable token, then the expression is relocatable. If the operand contains two or more relocatable tokens, then the expression is not relocatable.

Byte values are only relocatable candidates if the unary greater than **>** sign is used for the high byte and/or the unary less than **<** sign is used for the low byte. These operands are subject to the same relocation rules as full 16 bit address values.

Following are some examples illustrating these points.

```
LABEL1:     NOP                 ; The label is defined to be equal
                                  to the address of an instruction
                                  and therefore is relocatable.
LABEL2:     .EQUAL    LABEL1    ; The label is defined to equal a
                                  value that was tagged as reloc-
                                  atable. Therefore, LABEL2 is
                                  also relocatable.
LABEL3:     .EQUAL    10        ; The label is defined to equal a
                                  constant. Therefore, LABEL3 is
                                  not relocatable.
```

```
LABEL4:  .EQUAL    $+10   ; The label is defined to equal a relocatable
                          ; value plus a non-relocatable value. Since
                          ; only one value is relocatable, the symbol
                          ; LABEL4 is relocatable.

LABEL5:  .EQUAL    10+$   ; The label is defined to equal a non-relocatable
                          ; value plus a relocatable value. Since only one
                          ; value is relocatable, LABEL5 is relocatable.

LABEL6:  .EQUAL    LABEL5-LABEL2
                          ; The label is defined to equal a relocatable value
                          ; minus another relocatable value, producing a
                          ; non-relocatable result.
```

The last example is worth remembering when using the Assembler to do things such as calculate data sizes. Consider the following example of a table of data values, with the number of bytes being calculated automatically at assembly time by the Assembler, allowing the programmer to add or delete from the table without having to remember to change the data block size.

```
DATA:       .BYTE     0
            .WORD     10
            .BYTE     20
            .BLKB     5

DATA_SIZE:  .EQUAL    $-DATA
```

The Assembler will calculate the size of the data block, and because the result is not relocatable, the Linker will not alter the data block size.

Linker Examples

This section consists of examples intended to demonstrate the use of the Linker. The **<CR>** symbol denotes a carriage return and is shown only when no other response to a prompt is desired. Otherwise, all inputs are assumed to be terminated with a carriage return. In all cases, a Data File can be constructed with exactly the same responses as when running in Prompt mode.

Single File Assembled At Desired Run Address

The first example is the case of just one file which has been assembled at the desired run address by the use of the **ORIGIN** directive. Also, assume the default output file type is Executable, and no Linker options are desired. If no additional sections were defined, and there was no switching between the predefined sections, the Linker prompts would be as follows:

```
Input Filename : filename
        Enter Offset for 'CODE'          : 0
Input Filename : <CR>

Output Filename : <CR>

Library Filename: <CR>

Options (D, S, A, M, Z, X, H, E, T, 1, 2, 3, <CR>= Default ) : x
```

The above will cause the Linker to read the file **filename.obj**, add 0 to all relocatable addresses, and output a file with the name **filename.tsk**.

The following example shows the case where everything is the same as in the previous example except the desired output format is Intel Hex. Note that the default Linker output format may be changed with the **OPTIONS** Assembler directive.

```
Input Filename : filename
        Enter Offset for 'CODE'        : 0
Input Filename : <CR>

Output Filename :<CR>

Library Filename:<CR>

Options (D, S, A, M, Z, X, H, E, T, 1, 2, 3, <CR>= Default) : h
```

The last example for this type of file is the same as in the previous example with the addition of a disk Load Map file, user specified output filename, and a library to search for unresolved external references. The options may be specified in any order.

```
Input Filename : filename
        Enter Offset for 'CODE'        : 0
Input Filename :<CR>

Output Filename :user_filename

Library Filename: lib_filename

Options (D, S, A, M, Z, X, H, E, T, 1, 2, 3, <CR>= Default) : hd
```

Single File With Multiple Sections

This example demonstrates how the Linker handles multiple program sections. If the predefined **CODE** and **DATA** sections were used, and the **DATA** section is to be stacked on top of the **CODE** section, then the Linker prompts would be as follows:

```
Input Filename : filename
        Enter Offset for 'CODE'        : 0
        Enter Offset for 'DATA'        : <CR>
Input Filename : <CR>

Output Filename :<CR>

Library Filename: <CR>

Options (D, S, A, M, Z, X, H, E, T, 1, 2, 3, <CR>= Default) :
```

If instead of using the predefined sections **CODE** and **DATA**, the user defined sections **Program_section1** and **Program_section2** were used, and **Program_section2** is to be stacked on top of **Program_section1**, the prompts would be as follows:

```
Input Filename : filename
        Enter Offset for 'Program_section1'    : 0
        Enter Offset for 'Program_section2'    :<CR>
Input Filename : <CR>

Output Filename : <CR>

Library Filename: <CR>

Options (D, S, A, M, Z, X, H, E, T, 1, 2, 3, <CR>= Default) :
```

One important item to keep in mind is that sections are stacked in the order in which they are defined. Therefore, in this example there is no way to stack **Program_section2** on top of **Program_section1**. If the need arises to reverse the order, then order of the **SECTION** directives in the source file must be changed.

If in the above example, **Program_section1** was to be relocated to run at 2000H and **Program_section2** was to be relocated to run at 4000H, the following responses would be used.

```
Input Filename : filename
        Enter Offset for 'Program_section1'     : 2000
        Enter Offset for 'Program_section2'     : 4000
Input Filename : <CR>

Output Filename : <CR>

Library Filename: <CR>

Options (D, S, A, M, Z, X, H, E, T, 1, 2, 3, <CR>= Default) :
```

Note that the addresses are always specified in Hexadecimal. If the output file format was Executable, the gap from the end of **Program_section1** to the start of **Program_section2** would be filled in with the default fill character, which is FF Hex. This may be changed with the **FILLCHAR** Assembler directive.

Multiple Files With Multiple Sections

This example illustrates how the Linker handles section names in multiple files. Assume **file1** and **file2** use both **CODE** and **DATA** sections, and **file1** is to be linked starting at 0. The prompts will appear as follows:

```
Input  Filename : file1
               Enter Offset for 'CODE'        : 0
               Enter Offset for 'DATA'        : <CR>
Input  Filename : file2
               Enter Offset for 'CODE'        :<CR>
               Enter Offset for 'DATA'        : <CR>

Output Filename : <CR>

Library Filename: <CR>

Options (D, S, A, M, Z, X, H, E, T, 1, 2, 3, <CR>= Default) :
```

This will produce a file with both **CODE** sections stacked on top of each other, followed by both **DATA** sections being stacked on top of each other, then stacked on top of both **CODE** sections. This shows the general rule of stacking sections. Sections are stacked according to name, and are stacked in the order in which they are defined in the source file. All **CODE** sections will be grouped together, then all **DATA** sections, etc. **CODE** sections will always be stacked before **DATA** sections, since that is the order they are predefined in. If **DATA** must be placed before **CODE**, then **CODE** should not be used and a user defined section should be used. This is true for stacking only. If **CODE** and **DATA** are to be stacked, but placed at specific addresses, this would be done as follows, assuming **CODE** is to start at E000H and **DATA** is to start at 1000H.

```
Input  Filename : file1
                  Enter Offset for 'CODE'        : E000
                  Enter Offset for 'DATA'        : 1000
Input Filename :  file2
                  Enter Offset for 'CODE'        : <CR>
                  Enter Offset for 'DATA'        : <CR>

Output Filename : <CR>

Library Filename:<CR>

Options (D, S, A, M, Z, X, H, E, T, 1, 2, 3, <CR>= Default) :
```

The rules described in this example hold true regardless of how many input files there are. Sections can be used to separate program sections according to function, or to assist in complex linking, since any section may be placed at any address.

Single File With One Section Used For Reference Only

A reference only section is a section that is relocated so that any globals defined in the section can be used for linking purposes, however the section is not included in the output file. Reference only sections are useful in cases such as where the program resides in ROM or EPROM and the data areas reside in RAM. It is desirable to have the output file contain only that part of the program that is to be stored in ROM. Using an example along the same lines as the previous examples, assume that the program only uses the predefined **CODE** and **DATA** sections, that the **CODE** is to start at 1000H and the **DATA** is to be stacked on top of the **CODE**, used for linking purposes, and then discarded. A minus sign before a section address specifies that section as reference only. A minus sign before the section name indicates a reference only section in the Load Map.

```
Input  Filename : filename
               Enter Offset for 'CODE'        : 1000
               Enter Offset for 'DATA'        : -<CR>
Input  Filename : <CR>

Output Filename : <CR>

Library Filename: <CR>

Options (D, S, A, M, Z ,X, H, E, T, 1, 2, 3, <CR>= Default) :
```

If the **DATA** section was to be placed at 4000H instead of stacked on top of the **CODE** section, and was to be used for reference only, this could be accomplished as follows:

```
Input  Filename : filename
               Enter Offset for 'CODE'        :1000
               Enter Offset for 'DATA'        : -4000
Input  Filename : <CR>

Output Filename : <CR>

Library Filename: <CR>

Options (D, S, A, M, Z, X, H, E, T, 1, 2, 3, <CR>= Default) :
```

Any section of any file may be used for reference only with one exception. The first section in the first file may not be used for reference only, since it is used as the basis for all other Linker calculations. Therefore, it is a good idea not to use the **CODE** section for reference only. Instead, define a section with the **SECTION** Assembler directive, and make it reference only. If that section was named **Ref_only**, this would appear as follows:

```
Input Filename : filename
            Enter Offset for 'DATA'          : 1000
            Enter Offset for 'Ref_only'      : -4000
Input Filename : <CR>

Output Filename : <CR>

Library Filename: <CR>

Options (D, S, A, M, Z, X, H, E, T, 1, 2, 3, <CR>= Default) :
```

Indirect Linking

Indirect Linking is the term 2500 A.D. uses to describe a section of a file that is linked to run at an address other than the actual load address. This may be called phase and dephase also, however there is one major difference, namely that phase and dephase change the address at the assembly level, and indirect linking changes the address at link time.

This concept can be fairly confusing, and it is extremely easy to generate an output file that has the addresses so messed up it will never run. Despite this fact, there are times when this linking method is required.

Assume a single board controller application of some sort, where the program resides in ROM. If all of the data consists of lookup tables or constants, then there is no reason to move the data out of ROM, since it will never be written to. But if there is data that has been initialized to some value, but that value will change as the program runs, then that data must be moved from ROM to RAM by some sort of run time startup routine. If the address in RAM is the same as the address in ROM, then there is no problem, since the addresses generated by the Linker will be correct. However, in many cases the data that must be moved will simply be stacked on top of the previous section and burned into the ROM at that address. Now, it would be desirable to move this data to the same place every time, regardless of how the size of the other sections of the program change, and a likely candidate for this location would be either low or high RAM. The problem is, the Linker linked the data addresses to be where the data resides in the ROM, not where it is going to be moved to run in RAM.

Indirect Linking solves this problem. Any section of a file can be linked to run at an address other than where it resides in the load file (the ROM). The '@' sign before the load address is used to convey to the Linker that this is what is desired. All indirect addresses are automatically stacked on top of the previous section, or a section by the same name in a previous file.

Following is a list of rules that should be remembered when using the indirect linking feature.

Important Indirect Linking Rules

1) **Once a section name has been tagged as indirect, every indentical section name in following files will automatically be tagged as indirect also.**
2) **Indirect sections are stacked in the order they would be stacked in if they were not indirect.**
3) **Indirect sections cannot be reference only, since the whole point is to include the section in the load file.**

Assume that there are three files, named **file1.obj**, **file2.obj** and **file3.obj**, each of which have three sections , called **program**, **const_data** and **init_data**. If section **program** resides at 0, and the constant data section **const_data** is to be stacked on top of **program**, and section **init_data** is supposed to run at 1000H in RAM, but will be stored in ROM on power up, the following link procedure would be used.

```
Input Filename : file1
            Enter Offset for 'program'     : 0000
            Enter Offset for 'const_data'  : <cr>
            Enter Offset for 'init_data'   : @1000
Input Filename : file2
            Enter Offset for 'program'     : <cr>
            Enter Offset for 'const_data'  : <cr>
            Enter Offset for 'init_data'   : <cr>
Input Filename : file3
            Enter Offset for 'program'     : <cr>
            Enter Offset for 'const_data'  : <cr>
            Enter Offset for 'init_data'   : <cr>
Input Filename : <CR>

Output Filename : <CR>

Library Filename: <CR>

Options (D, S, A, M, Z, X, H, E, T, 1, 2, 3, <CR>= Default) : <cr>
```

The resulting output file will contain all the sections, with **init_data** stacked on top of **const_data**, and both of these sections stacked on top of **program**. The Load Map will show the actual load addresses, however a look at the global symbol list will show that all global symbols defined in **init_data** are linked starting at 1000H. Therefore, the program will not run as is. The **init_data** section must be

moved to location 1000H. In order to do this, the size of the **init_data** section must be known. The easiest and most versatile way of finding the size of a section is to bracket it with other sections, even if they are empty. The Linker will not prompt for the load address of an empty section, but will if there is at least one equate in it. Therefore, the first file can be used to set the section link order. The following lines of code would be put in **file1.asm** to do this.

```
program:              .section
program_addr:         .equal      $
const_data:           .section
const_data_addr:      .equal      $
const_data_end:       .section
const_data_end_addr:  .equal      $
init_data:            .section
init_data_addr:       .equal      $
init_data_end:        .section
init_data_end_addr:   .equal      $

                      program
```

There are two sections in the above example that have no purpose other than address calculation. The section **const_data_end** provides the address in the ROM where **init_data** starts. The **init_data** section cannot be used directly because it has been linked at a different address, so all labels associated with **init_data** have been relocated with respect to that address. The section **init_data_end** provides the size of the **init_data** section. So, the following will provide the necessary information:

```
Size of Initialized Data = init_data_end_addr-init_data_addr
Address in Rom of Initialized Data = const_data_end_addr
```

Since the labels used to calculate the size of the initialized data section reside in different sections, the subtraction to calculate the size of the initialized data section must be performed by the run time startup routine.

The procedure to link the files together would be to link the files as shown in the previous example, with the addition of stacking **const_data_end** on top of

const_data and stacking **init_data_end** on top of **init_data.** In order for the size of **init_data** to come out correctly, the section **init_data_end** must be linked as indirect and stacked. This can be done by specifying '@' followed by a carriage return, as shown in the following example.

```
Input Filename : file1
        Enter Offset for 'program'          : 0000
        Enter Offset for 'const_data'       : <cr>
        Enter Offset for 'const_data_end'   : <cr>
        Enter Offset for 'init_data'        :1000
        Enter Offset for 'init_data_end'    : @<cr>
Input Filename : file2
        Enter Offset for 'program'          : <cr>
        Enter Offset for 'const_data'       : <cr>
        Enter Offset for 'const_data_end'   : <cr>
        Enter Offset for 'init_data'        : <cr>
        Enter Offset for 'init_data_end'    : <cr>
Input Filename : file3
        Enter Offset for 'program'          : <cr>
        Enter Offset for 'const_data'       : <cr>
        Enter Offset for 'const_data_end'   : <cr>
        Enter Offset for 'init_data'        : <cr>
        Enter Offset for 'init_data_end'    : <cr>
Input Filename : <CR>

Output Filename : <CR>

Library Filename: <CR>

Options (D, S, A, M, Z, X, H, E, T, 1, 2, 3, <CR>= Default) : <cr>
```

Since a file that is linked using indirect sections will usually have all sections stacked on top of the previous one, the linker can be put in auto-stack mode by specifying a semi-colon (;) after any of the load addresses, or before any of the carriage return only responses. After this, all sections will automatically be stacked.

In the example above, the runtime address specified were between 0000 and FFFFH. The addresses may also be speified in a segment and offset format. A runtime offset specified as a segment and offset are mainly used for the 8086, Z8000 and the 65816 micrprocessors. The output format selected should also be Extended Intel Hex. The example below shows program_section1 relocated at 1:0000H and program_section2 relocated to run at 1:2000H.

```
Input Filenme : filename
        Enter Offset for 'program_section1' : 1:0000
        Enter Offset for 'program_section2' : 2:0000
Input Filename: <cr>

Output Filename: <cr>

Library Filename: <cr>

Options (D, S, A, M, Z, X, E, T, 1, 2, 3, <CR> = Default): E
```

Note that the Extended Intel Hex option was chosen in the above example.

Linker Symbol Table Output Formats

The following sections describe the Symbol Table output formats from the Linker when one of the Linker Symbol Table options are chosen.

Symbol Table Output Format

This section describes the format of the Global Symbol Table that is produced when the **S** Linker option is selected. The Symbol Table always receives the same filename as the Linker output filename, with an extension of **SYM**. The first byte of this file is the i.d. code, which is E0H. The following bytes, relative to the start of each entry, are repeated for each entry.

Bytes	0-31	Global Symbol Name. The name is padded with zeros to fill out the 32 character positions. The end of the entries can be detected by an FFH in byte 0.
Byte	32	Most significant byte of relocated Global Symbol value.
Byte	33	Least significant byte of relocated Globol Symbol value.
Byte	34	File number in which the Global was defined. This is used by the linker to output the filename along with the value. This byte may be deleted or used for other purposes if desired.
Byte	35	Flag byte. This byte is unused at the current time but may be used in the future by 2500 A.D.

Abbreviated Global Symbol Table Output Format

This section describes the format of the Global Symbol Table that is produced when the **A** Linker option is selected. The Symbol Table always receives the same filename as the Linker output filename, with an extension of **SYM**. This is the same symbol table as produced by the 3.0 series of 2500 A.D. Linkers.

Bytes	0 - 9	Global Symbol Name. The name is padded with zeros to fill out the 10 character positions. The end of the entries can be detected by an FFH in byte 0.
Byte	10	Most significant byte of relocated Global Symbol value.
Byte	11	Least significant byte of relocated Global Symbol value
Byte	12	File number in which the Global was defined. This is used by the Linker to output the file-name along with the value.This byte may deleted or used for other purposes if desired.
Byte	13	Flag byte. This byte is unused at the current time but may be used in the future by 2500 A.D. products.

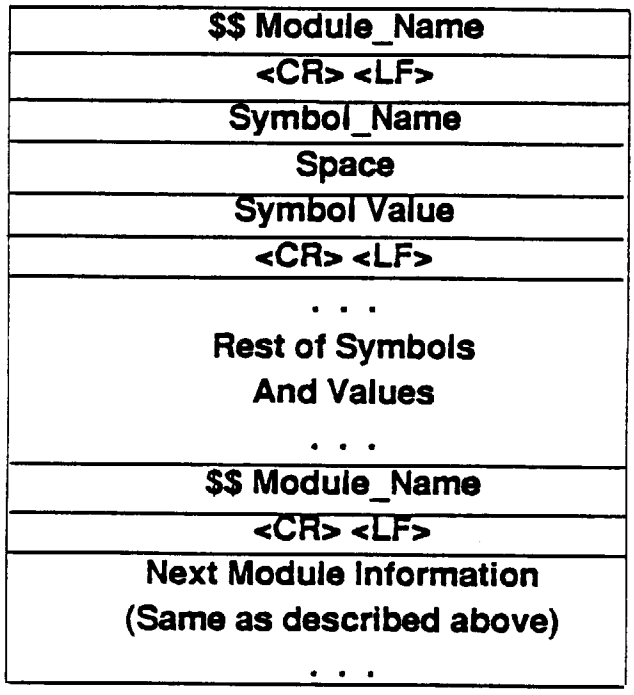
Microtek Symbol Table Output Format

This section describes the MicroTek Symbol Table format which is selected by the **M** Linker option. The Symbol Table always receives the same filename as the Linker output filename, with an extension of **SYM.**

FEH	Start of Module
Size of Module Name	
Module Name	
Rest of Module Name	3 Bytes in Length
Size of Symbol Address	2 = 16 Bits 3 = 24 Bits 4 = 8086,80186,80286 5 = 32 bits
Size of Symbol	1 Byte in Length
Symbol Name	
High Byte of Address	
Low Byte of Address	
. . . **Rest of Symbols & Values** . . .	
FEH	End of Module
Next Module Information	
(Same as described above)	
FFH	End of File

Zax Symbol Table Output Format

This section describes the format of the Zax Symbol Table that is produced when the **Z** Linker option is selected. The Symbol Table always receives the same filename as the Linker output filename with an extension of **.SYM.**



Start of Module

End of Module

Linker Output Formats

The following sections describe the output formats from the Linker when one of the Linker output formats are chosen

Intel Hex Format

The Intel Hex Format is described below.

Record Mark Field - This field signifies the start of a record, and consists of an Ascii colon (:).

Record Length Field - This field consists of two Ascii characters which indicate the number of data bytes in this record. The characters are the result of converting the number of data bytes in binary to two Ascii characters, high digit first. An end of file record contains two Ascii zeros in this field. The maximim number of data bytes in a record is 255. This can be changed by using the RECSIZE directive.

Load Address Field - This field consists of the four Ascii characters which result from converting the binary value of the address in which to begin loading this record. The order is as follows:

High digit of high byte of address.
Low digit of high byte of address
High digit of low byte of address.
Low digit of low byte of address.

In an end of file record, this field consists of either four Ascii zeros, or the program entry address.

Record Type Field - This field identifies the record type, which is either 00 for data records or 01 for an end of file record. It consists of two Ascii characters, with the high digit of the record type first, followed by the low digit of the record type.

Data Field - This field consists of the actual data, converted to two Ascii characters, high digit first. There are no data bytes in the end of file record.

Checksum Field - The checksum field is the 8 bit binary sum of the record length field, the load address field, the record type field and the data field. This sum is then negated (2's complement) and converted to two Ascii characters, high digit first.

Extended Intel Hex Format

The Extended Intel Hex Format is described below.

Record Mark Field - This field signifies the start of a record, and consists of an Ascii colon (:).

Record Length Field - This field consists of two Ascii characters which indicate the number of data bytes in this record. The characters are the result of converting the number of data bytes in binary to two Ascii characters, high digit first. An end of file record contains two Ascii zeros in this field. The maximim number of data bytes in a record is 255. This can be changed by using the RECSIZE directive.

Load Address Field - This field consists of the four Ascii characters which result from converting the binary value of the address in which to begin loading this record. The order is as follows:

High digit of high byte of address.
Low digit of high byte of address
High digit of low byte of address.
Low digit of low byte of address.

In an end of file record, this field consists of either four Ascii zeros, or the program entry address.

Record Type Field - This field identifies the record type, which is either 00 for data records or 01 for an end of file record and 02 for a segment address . It consists of two Ascii characters, with the high digit of the record type first, followed by the low digit of the record type.

Data Field - This field consists of the actual data, converted to two Ascii characters, high digit first. There are no data bytes in the end of file record.

Checksum Field - The checksum field is the 8 bit binary sum of the record length field, the load address field, the record type field and the data field. This sum is then negated (2's complement) and converted to two Ascii characters, high digit first.

Motorola S19 Format

The Motorola S1 - S9 Format is described below.

Record Type Field - This field signifies the start of a record and the record type as follows:

Ascii S1 - Data Record
Ascii S9 - End of File Record

Record Length Field - This field specifies the record length which includes the Address, Data and Checksum fields. The 8 bit Record Length value is converted to two Ascii characters, high digit first. Since the smallest object file record size is 128 bytes, the Record Length field always consists of 128 data bytes, 2 address bytes and 1 checksum byte, resulting in a record length of 131 bytes. This can be changed with the RECSIZE directive.

Load Address Field - This field consists of the four Ascii characters which result from converting the binary value of the address in which to begin loading this record. The order is as follows:

High digit of high byte of address
Low digit of high byte of address
High digit of low byte of address
Low digit of low byte of address

In an end of file record, this field consists of four Ascii zeros.

Data Field - This field consists of the actual data, converted to two Ascii characters, high digit first. There are no data bytes in an end of file record.

Checksum Field - The checksum field is the 8 bit binary sum of the record length field, the load address field and the data field. This sum is then complemented (1's complement) and converted to two Ascii characters, high digit first.

Motorola S28 Format

The Motorola S2 - S8 Format is described below.

Record Type Field - This field signifies the start of a record and identifies the record type as follows:

Ascii S2 - Data Record
Ascii S8 - End of File Record

Record Length Field - This field specifies the record length which includes the Address, Data and Checksum fields. The 8 bit Record Length value is converted to two Ascii characters, high digit first.

Load Address Field - This field consists of the six Ascii characters which result from converting the binary value of the address in which to begin loading this record. The order is as follows:

High digit of high byte of address
Low digit of high byte of address
High digit of mid byte of address
Low digit of mid byte of address
High digit of low byte of address
Low digit of low byte of address

In an end of file record, this field consists of six Ascii zeros.

Data Field - This field consists of the actual data, converted to two Ascii characters, high digit first. There are no data bytes in an end of file record.

Checksum Field - The checksum field is the 8 bit binary sum of the record length field, the load address field and the data field. This sum is complemented (1's complement) and converted to two Ascii characters, high digit first.

Motorola S37 Format

The Motorola S3 - S7 Format is described below.

Record Type Field - This field signifies the start of a record and identifies the record type as follows:

Ascii S3 - Data Record
Ascii S7 - End of File Record

Record Length Field - This field specifies the record length which includes the Address, Data and Checksum fields. The 8 bit Record Length value is converted to two Ascii characters, high digit first.

Load Address Field - This field consists of the eight Ascii characters which result from converting the binary value of the address in which to begin loading this record. The order is as follows:

High digit of high byte of high word
Low digit of high byte of high word
High digit of low byte of high word
Low digit of low byte of high word
High digit of high byte of low word
Low digit of high byte of low word
High digit of low byte of low word
Low digit of low byte of low word

In an end of file record, this field consists of eight Ascii zeros.

Data Field - This field consists of the actual data, converted to two Ascii characters, high digit first. There are no data bytes in an end of file record.

Checksum Field - The checksum field is the 8 bit binary sum of the record length field, the load address field and the data field. This sum is complemented (1's complement) and converted to two Ascii characters, high digit first.

2500 A.D. Librarian

2500 A.D. Librarian Description

The 2500 A.D. Librarian is used to create a library of user specified object modules to be linked together with a program created by a 2500 A.D. assembler. The linker will search the specified libraries and only include the referenced library modules.

The Librarian will process any size file, as long as enough disk space is available. A library is limited to **256** separate modules. The Librarian requires enough disk space for an existing library and space for a temporary file the size of the existing library plus any modules being added to the library. A temporary file is used to create a library to minimize the possibility of damage to an existing library. The Librarian also requires enough memory to store all the global symbol records of the library modules, and the module directory list. The global symbols are checked for multiply defined symbols.

The Librarian command line parser only recognizes two operand separators space and tab. The command descriptions show the full command name and the allowed abbreviations. The operands allowed by each command are also shown. The system defaults used by the Librarian are :

obj	object filename extension
pak	packed object filename extension
lib	library filename extension
tmp	temporary filename extension

The Librarian displays the modules contained in a library on the screen. The current working module is displayed with highlighting. The current working module can be changed by scrolling up or down the library directory list. To scroll up the directory list type **k** or **K** and to scroll down the directory list type **j** or **J**. The directory list can be displayed 16 modules at a time. If a module is added to the list and is outside the currently displayed modules, the screen is adjusted to display the added module. The added module is always displayed as the current module. A

2500 A.D. Librarian

module may be a single object file or a module in a packed object file. A packed object file is several object files concatenated together. See the **MODULE** and **ENDMOD** directive descriptions in the Assembly Mode section of this manual for examples of creating a packed object file. A packed object file may be used for an operation on a single module or all modules within the file. The operand **all** or **ALL** may only be used with a packed object file, and will cause all the modules within the file to be added to a library or replace the modules in the library.

Note to VMS users:

VMS users must type carriage return after using **'j'** or **'J'** as well as **'k'** or **'K'**.

2500 A.D. Librarian

Librarian Installation

The 2500AD Librarian is included on the distribution media for the Assembler. The filename for the librarian on the UNIX or ULTRIX operating system is LIB. The filename for the librarian on the MSDOS or VMS operating system is LIB.EXE. Installation of the librarian is dependent on the host operating system. Users of the UNIX or ULTRIX operating systems must perform step 1 of the installation directions. Users of the MSDOS or VMS operating systems must also perform the appropriate section of step 2 of the installation directions.

STEP 1:	**Copy the file lib or lib.exe to the desired directory and disk drive.**

NOTE: The file LIB.EXE will need to be renamed on the VMS operating system so the 2500AD librarian will not conflict with the VMS librarian utility.

STEP 2:	**This only applies to users of the MSDOS or VMS operating systems. Please refer to the section appropriate to your host operating system.**

MSDOS:

The librarian requires the use of the ANSI device driver. The systems **config.sys** file must be modified to cause the MSDOS operating system to load the device driver. The config.sys file is located in the root directory. if the **config.sys** file does not exits one must be created. Please refer to your MSDOS manual for information on creating a **config.sys** file. Add the following line to the **config.sys** file.

```
DEVICE=ANSI.SYS
```

NOTE: The system will need to be rebooted for the MSDOS operating system to load the ANSI device driver.

2500 A.D. Librarian

VMS:

The librarian will need to be defined as a command to the VMS operating system, if you wish to execute the librarian without using the RUN command. The following line must be added to the LOGIN.COM file.

librarian == "$$Disk1:librarian.exe"

The filename for the librarian shown in the previous line is dependent upon the name the librarian was given when it was copied from the distribution media.

2500 A.D. Librarian

Librarian Operating Instructions

The Librarian will accept one command line argument, the filename of an existing library or a library to be created.

To run the Librarian and read an existing library type :

```
lib filename
```

Where the filename is an existing library.

To run the Librarian and create a new library type :

```
lib filename
```

Where the filename is the new library name.

The Librarian may also be invoked with no command line arguments and an existing library or a new library specified from the Librarian command line.

To run the Librarian with no file specified type :

```
lib
```

The Librarian will respond with the prompt :

Enter Command :

The Librarian commands will be described in the next section. The Librarian is designed to be used interactively and cannot be used with an Msdos batch file or Unix shell file.

2500 A.D. Librarian

ADD filename
A

The **ADD** command adds a module to a library, or replaces an existing module in a library. The command accepts one operand, the filename of a module to be added to a library or the operand **all** or **ALL**. The operand **all** or **ALL** is only allowed with a packed object file. The **ADD** command will not allow a module to be added unless a library has been open or created with the **NEW** command. The Librarian will always prompt for the object filename containing the module to be added. If the file specified is a packed object file the file is searched for the module, and if found the module is added to the library. If the maximum module limit is reached the module will not be added. If the operand is not present the Librarian will prompt for the module name.

Command Examples:

```
add printf
```

The Librarian will respond with the prompt :

Enter Name of Object File :

Assume the name mylib was entered at the prompt. The Librarian will search for the file mylib.obj, and if the file is not found the Librarian will search for the file mylib.pak.

```
add all
```

The Librarian will respond with the prompt :

Enter Name of Object File :

2500 A.D. Librarian

Assume the filename is mylib.pak. The Librarian will add all the modules contained in the file mylib.pak to the library.

```
add
```

The Librarian will respond with the prompt :

Enter Name of Module :

After the module name is entered the Librarian will prompt for the name of the file containing the module as in the previous examples.

2500 A.D. Librarian

DEL module
D

The **DEL** command deletes the named module or the current working module from the library. The current working module is the highlighted module on the screen. If the operand is not present the current working module is deleted. If the operand is present the library directory list is searched for the named module. It is an error to delete a module if no library has been opened, the library contains no modules or the named module cannot be found in the directory list.

Command Examples:

```
del printf
```

The Librarian will search for the directory entry printf and delete the entry from the list removing the module from the library. The next module in the directory list becomes the current working module, and if the deleted module was the last module in the list the previous module becomes the current working module.

```
del
```

The Librarian will delete the current working module from the directory list removing the module from the library.

2500 A.D. Librarian

EXIT

The **EXIT** command will save a library, if a library is open and contains at least one module, and return control to the operating system.

Command Examples:

```
exit
```

If a library is open and contains at least one module the Librarian will display the message :

Saving Library : library name

FIND module

The **FIND** command may be used to locate a module in a large library. The command accepts one operand and the name of the module to find. The module, if found, will become the current or highlighted module. An error message will be displayed if the module cannot be found in the library.

Command Example:

```
find printf
```

HELP command name
H

The **HELP** command displays a summary of all the Librarian commands, or a description of the command specified by the optional operand. The directory display is cleared and the help text is displayed in place of the directory list. The Librarian prompts for any character to be typed to continue. The directory list is restored and the Librarian will wait for the next command.

Command Example:

```
HELP
```

The main help screen describing how to use the **HELP** command is displayed. The Librarian will respond with the prompt :

Press Any Key To Continue :

2500 A.D. Librarian

```
help ADD
```

The help screen describing how to use the **ADD** command is displayed and the Librarian responds with the prompt :

Press Any Key To Continue :

LIST .module

The list command will list the global symbols of the specified module to the screen or to a file. The global symbols are listed by name in sorted order. The command may have two operands. The first operand the module name or the word ALL or all. The operand ALL will list the global symbols for all modules in the library. The second operand is optional and must be the word disk or DISK. The second operand specifies sending the global symbol listing to a disk file .

Command Examples:

```
LIST mod2
LIST ALL disk
```

2500 A.D. Librarian

NEW filename
N

The **NEW** command creates a new library or opens an existing library. If a library was previously open and contained at least one module the previous library will be saved before the new library is opened. The command requires one operand the filename of the library to create or open; if the operand is not present the Librarian will prompt for the filename. A library must be open before modules can be added, deleted or replaced.

Command Example:

```
new mylib
```

The Librarian will search for an existing library with the filename mylib.lib. If the library exists the library directory is displayed on the screen. If the library does not exist a library with the name mylib.lib will be created.

QUIT

The **QUIT** command causes the Librarian to terminate and return control to the operating system. If a library is open and contains at least one module the Librarian will ask whether the library is to be saved. Typing **y** or **Y** will cause the library to be saved, any other character will result in the library being abandoned.

Command Example:

```
QUIT
```

If a library is open and contains at least one module the Librarian will respond with the prompt :

Do You Want to Save the Library (y/n) :

Any character other than **y** or **Y** will cause the library to abandoned.

2500 A.D. Librarian

REP module
R

The **REP** command is used to replace an existing module in a library with a new version of the module. The command accepts one optional operand the module to replace, or the operand **all** or **ALL**. The **all** operand may only be used with packed object files. If the operand is present the library directory list is searched for the named module, and if the module is found it is replaced. If the module is not found an error message is displayed. If the operand is not present the current working module is replaced. The current working module is defined as the module highlighted on the screen. It is an error to replace a module if no library is open or the library contains no modules. The Librarian will always prompt for the object filename containing the updated version of the module to be replaced.

Command Examples:

```
rep printf
```

The Librarian will search the library directory list for the module printf, and if the module is found displays the prompt :

Enter Name of Object File :

Assume the filename mylib was entered. The Librarian will first search for the file mylib.obj. If the file mylib.obj is not found the Librarian will search for the file mylib.pak, and if the file is found search for the module printf.

```
rep
```

The Librarian will use the module name of the current working module (highlighted module) as the module to replace. The Librarian then displays the prompt :

Enter Name of Object File :

2500 A.D. Librarian

STAT
S

The **STAT** command displays the status of the current library. If no library is open the command displays a message that no library is open. If a library is open the total number of modules, externals, globals, and the library name is displayed. The directory display is overwritten and the status message is displayed, the Librarian then prompts for any key to be pressed to continue. The directory display is then restored.

Command Example:

```
stat
```

The library status is displayed, and the Librarian displays the prompt:

Press Any Key To Continue :

After a key is pressed the directory list display is restored and the Librarian waits for the next command to be entered.

2500 A.D. Librarian

TOP

The **TOP** command moves the current working directory highlighting to the first module in the directory list.

Command Example:

```
top
```

BOT

The **BOT** command moves the current working directory highlighting to the last module in the directory list.

Command Example:

```
bot
```

2500 A.D. Librarian

Librarian Error Messages

This Section provides a list of the error messages output by the Librarian and an explanation of the error message. The command line is output with the error message unless otherwise specified.

Error - Input Line Too Long
Meaning - The input line exceeded 80 characters.

Error - Illegal Command
Meaning - The command entered was not a valid command or the ADD,DEL or REP command was used and a library was not open.

Error - Read Error
Meaning - An error occurred while a file was being read. The name of the file is output with the error message.

Error - Write Error
Meaning - An error occurred while a file was being updated. The name of the file is output with the error message.

Error - Cannot Open File
Meaning - The file specified could not be accessed or opened. The name of the file is output with the error message.

Error - Multiple Defined Module
Meaning - The module name being added to a library already exists in the library. Replace the module or rename the module to be added to the library. The name of the file is output with the error message.

Error - Filename Too Long
Meaning - The filename specified was too long after the default extension was appended.

2500 A.D. Librarian

Error - Undefined Module.
Meaning - The module being replaced or deleted from library does not exist in the library

Error - Not Enough memory
Meaning - There was not enough memory for the buffer or symbol table space required

Error - Seek Error
Meaning - An error occurred while the position within a file was being updated.

Error - Cannot Delete Old Library
Meaning - An old library file could not be deleted after the new library was created or updated. The name of the file is output with the error message.

Error - Cannot Create New Library
Meaning - The temporary library file could not be renamed. The name of the file is output with the error message.

Error - Incompatible Object Module
Meaning - A library file was specified with the ADD command or an object module was specified with the NEW command.

Error - Too Many Command Line Arguments
Meaning - The Librarian was invoked with more than one argument from the operating system command line.

Error - Multiple Defined Global Symbol
Meaning - A global symbol in a module being added to a library contains a symbol name that exists in another module.The symbol name and the filename are output with the error message.

Error - Maximum Module Count Exceeded
Meaning - The maximum module count of 256 modules was exceeded.

Error - Must Be A Packed Object File
Meaning - The operand "all" or "ALL" was used for the ADD or REP commands, and the file specified was not a packed object file.

2500AD Software System Requirements

System Requirements

For **MSDOS** systems, the minimum system requirement is **512k** of available memory, minus the amount used by the operating system. Unix systems require at least 1 megabyte of memory.

For **MSDOS** systems, up to 20,000 lines of source code have been assembled with 512k of memory, and 30,000 line programs have been assembled with 640k of memory without running out of space.

The Linker assumes **20 files** may be open at once. On **MSDOS** systems there is a file called **CONFIG.SYS** that is on the system disk. The default value is 5 files, therefore, this must be changed to at least 20. Since the Linker will open and close files when the number exceeds 20, raising the number higher than 20 will have no effect on the Linker. However, if any memory resident programs open files, the number should be increased by the number of files used by the memory resident programs.

8080 To Z80 Source Code Converter

The 2500 A.D. 8080 to Z80 Source Code Converter will convert standard Intel 8080 source code to Zilog source code, which can be subsequently assembled on the 2500 A.D. Z80 Cross Assembler. Since the 8080 source buffer, Z80 source buffer and symbol table buffer overflow to disk, the size of any file that can be converted is limited only by the available disk storage space.

Is should be noted that this only runs under Msdos and Zeus.

To run the Converter type : CONV

The Converter will respond with the following prompt.

INPUT FILENAME ? :

After the user enters the 8080 source code filename the Converter will ask for the Z80 output filename as shown below.

OUTPUT FILENAME ? :

The Converter will display the 8080 source line, followed by the Z80 equivalent at the terminal. Any source lines which the Converter does not recognize are assumed to be macros. Although only macros will normally fall into this class, it may be possible to actually define macros in the Z80 file to handle some of the non-standard Z80 instruction extensions added to existing 8080 assemblers.

Part of the conversion process includes building a symbol table. This table is deleted at the end of a normal converter run, but if for some reason the conversion is aborted, a file of the name **SYMBOL.CON** may be left on the disk and should be deleted by the operator.

The Converter recognizes two types of comment lines. The first is a single line with a semi-colon in column 1. For large comment blocks, some assemblers have a special assembler directive. The Converter will recognize a comment block as follows:

```
.COMMENT X
```

where **X** can be any character. The Converter will treat everything from the first **X** to the second **X** as a comment block.

Ascii Chart

ASCII CHART

Character	Binary	Octal	Decimal	Hex
NUL	00000000	000	000	00
SOH	00000001	001	001	01
STX	00000010	002	002	02
ETX	00000011	003	003	03
EOT	00000100	004	004	04
ENQ	00000101	005	005	05
ACK	00000110	006	006	06
BEL	00000111	007	007	07
BS	00001000	010	008	08
HT	00001001	011	009	09
LF	00001010	012	010	0A
VT	00001011	013	011	0B
FF	00001100	014	012	0C
CR	00001101	015	013	0D
SO	00001110	016	014	0E
SI	00001111	017	015	0F
DLE	00010000	020	016	10
DC1	00010001	021	017	11
DC2	00010010	022	018	12
DC3	00010011	023	019	13
DC4	00010100	024	020	14
NAK	00010101	025	021	15
SYN	00010110	026	022	16
ETB	00010111	027	023	17
CAN	00011000	030	024	18
EM	00011001	031	025	19
SUB	00011010	032	026	1A
ESC	00011011	033	027	1B
FS	00011100	034	028	1C
GS	00011101	035	029	1D
RS	00011110	036	030	1E
US	00011111	037	031	1F
SP	00100000	040	032	20
!	00100001	041	033	21
"	00100010	042	034	22
#	00100011	043	035	23
$	00100100	044	036	24
%	00100101	045	037	25
&	00100110	046	038	26
'	00100111	047	039	27
(	00101000	050	040	28
)	00101001	051	041	29

Ascii Chart

Character	Binary	Octal	Decimal	Hex
*	00101010	052	042	2A
+	00101011	053	043	2B
,	00101100	054	044	2C
-	00101101	055	045	2D
.	00101110	056	046	2E
/	00101111	057	047	2F
0	00110000	060	048	30
1	00110001	061	049	31
2	00110010	062	050	32
3	00110011	063	051	33
4	00110100	064	052	34
5	00110101	065	053	35
6	00110110	066	054	36
7	00110111	067	055	37
8	00111000	070	056	38
9	00111001	071	057	39
:	00111010	072	058	3A
;	00111011	073	059	3B
<	00111100	074	060	3C
=	00111101	075	061	3D
>	00111110	076	062	3E
?	00111111	077	063	3F
@	01000000	100	064	40
A	01000001	101	065	41
B	01000010	102	066	42
C	01000011	103	067	43
D	01000100	104	068	44
E	01000101	105	069	45
F	01000110	106	070	46
G	01000111	107	071	47
H	01001000	110	072	48
I	01001001	111	073	49
J	01001010	112	074	4A
K	01001011	113	075	4B
L	01001100	114	076	4C
M	01001101	115	077	4D
N	01001110	116	078	4E
O	01001111	117	079	4F
P	01010000	120	080	50
Q	01010001	121	081	51
R	01010010	122	082	52
S	01010011	123	083	53
T	01010100	124	084	54

Ascii Chart

Character	Binary	Octal	Decimal	Hex
U	01010101	125	085	55
V	01010110	126	086	56
W	01010111	127	087	57
X	01011000	130	088	58
Y	01011001	131	089	59
Z	01011010	132	090	5A
[	01011011	133	091	5B
\	01011100	134	092	5C
]	01011101	135	093	5D
^	01011110	136	094	5E
_	01011111	137	095	5F
`	01100000	140	096	60
a	01100001	141	097	61
b	01100010	142	098	62
c	01100011	143	099	63
d	01100100	144	100	64
e	01100101	145	101	65
f	01100110	146	102	66
g	01100111	147	103	67
h	01101000	150	104	68
i	01101001	151	105	69
j	01101010	152	106	6A
k	01101011	153	107	6B
l	01101100	154	108	6C
m	01101101	155	109	6D
n	01101110	156	110	6E
o	01101111	157	111	6F
p	01110000	160	112	70
q	01110001	161	113	71
r	01110010	162	114	72
s	01110011	163	115	73
t	01110100	164	116	74
u	01110101	165	117	75
v	01110110	166	118	76
w	01110111	167	119	77
x	01111000	170	120	78
y	01111001	171	121	79
z	01111010	172	122	7A
{	01111011	173	123	7B
|	01111100	174	124	7C
}	01111101	175	125	7D
~	01111110	176	126	7E
DEL	01111111	177	127	7F

Ascii Chart

Abbreviations for Control Characters

NUL	-	null, or all zeros
SOH	-	start of heading
STX	-	start of text
ETX	-	end of text
EOT	-	end of transmission
ENQ	-	enquiry
ACK	-	acknowledge
BEL	-	bell
BS	-	backspace
HT	-	horizontal tabulation
LF	-	line feed
VT	-	vertical tabulation
FF	-	form feed
CR	-	carriage return
SO	-	shift out
SI	-	shift in
DLE	-	data link escape
DC1	-	device control 1
DC2	-	device control 2
DC3	-	device control 3
DC4	-	device control 4
NAK	-	negative acknowledge
SYN	-	synchronous idle
ETB	-	end of transmission block
CAN	-	cancel
EM	-	end of medium
SUB	-	substitute
ESC	-	escape
FS	-	file separator
GS	-	group separator
RS	-	record separator
US	-	unit separator
SP	-	space
DEL	-	delete

Index

V

Z