CHAPTER 11

EDIT-ASM-LINK-XBUG

This chapter explains the use of the PDOS development software tools EDIT, ASM, LINK, and XBUG. Assembly and BASIC applications are written, assembled, linked, and debugged using these utility programs.

Included with PDOS are three editors: a virtual screen editor (JED), a non-virtual screen editor (JEDY), and a small character editor (EDIT). The 9900 assembler (ASM), the module linker (LINK), and the resident debugger (XBUG) are also necessary for application development.

11.1 EDITORS

11.1.1 JED - VIRTUAL SCREEN EDITOR

JED is a screen oriented editor designed for terminals with 24 x 80 character displays and cursor addressing. JED features numerous text editing capabilities in REPLACE and INSERT modes. All character editing is immediately displayed on the screen; the screen always reflects the true image of the text being edited. Consequently, you are not likely to become confused or lost as is the case with character and line editors.

Screen oriented editor

REPLACE and INSERT modes

The first twenty-three lines of the display are used to window into the editor text buffer. The twenty-fourth line displays prompt and status messages associated with JED commands. The cursor indicates the place at which at transactions take place, including character insertion, deletion, and replacement. In INSERT mode, text is inserted by simply typing the desired characters. The text is adjusted to the right and below automatically. In REPLACE mode, buffer characters are overwritten as new characters are entered. In both modes, control keys are used to invoke special editing functions.

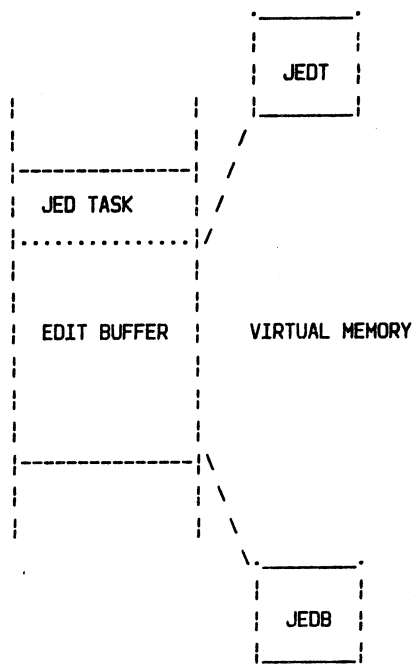
23 line window
1 status line

INSERT mode

REPLACE mode

JED is a virtual editor, not limited to just the available memory space for editing large files. If the edit buffer exceeds user task space, JED moves text to and from temporary disk files. The management of the temporary files is transparent to the user. Two files are maintained on the system disk for this purpose. They are defined automatically when JED begins execution.

Virtual editor



JED has a priority structure for processing editor functions. JED commands have the highest priority and override all other functions. The screen update processor has second priority and updates only those characters which change on the screen. After each command, this process is restarted from the top of the screen. The lowest priority task is a time-of-day clock in the lower right hand corner of the screen. The clock is updated only when JED is idle.

(11.1.1 JED - VIRTUAL SCREEN EDITOR continued)

Control character commands

Cursor movement and other special commands are initiated by control characters. (Control characters are indicated by a "^" symbol preceding a character.) Some terminals already have function keys that can be used directly for cursor movement. The escape key is a special control function which is used to delimit multiple key commands and set JED modes.

24th status line

Certain commands require only the control character; others require a string of characters, such as a file name. JED prompts on the 24th line for those requiring further information. When the prompt is given, the string is entered and then terminated with an <esc>. If a mistake is made while typing the string, then the <rubout> key erases the last character entered. The command is aborted by entering a ^C or by deleting the whole string with <rubout> and entering an <esc>.

Last command recall

The last command string for GET FILE, WRITE FILE, and SEARCH commands is recalled by repeating the command key twice. Two recall buffers are maintained, one for the GET <^G> and WRITE <^W> FILE commands, and the other for the SEARCH <^S> and <^B> commands. Thus, to write an edited file back to the original file, a <^W> <^W> <esc> <V> sequence prompts for WRITE FILE, recalls the file name, delimits the command, and verifies the action. Likewise, the last search command is repeated by entering <^S> <^S> <esc>.

Operator verification

All file operations by the JED editor require an operator verification. This is done by JED prompting with a 'VERIFY' in the lower right hand corner of the screen after the <esc> key has been entered. A <V> or <v> will change the 'VERIFY' to 'VERIFIED' and the command is executed. Any other character changes 'VERIFY' to 'NOT VERIFIED' and the command is aborted.

Auto-create

If the output file is not defined, JED prompts with 'CREATE VERIFY'. A <V> or <v> key changes the prompt to 'CREATE VERIFIED', defines the new file, and writes to that file. This applies to OUTPUT BLOCK and WRITE FILE commands.

(11.1.1 JED - VIRTUAL SCREEN EDITOR continued)

JED COMMANDS

<^A> INSERT FROM UP BUFFER. The INSERT FROM UP BUFFER command inserts text from the 'UP' buffer, beginning at the cursor. (See <^U>.)

INSERT FROM UP BUFFER

<^B> BACKWARD SEARCH. The BACKWARD SEARCH initiates a text search, starting at the cursor to the beginning of the text buffer. The command prompts with "- Search for '" after which the text string is entered. If the string is found, the cursor is positioned on the leftmost character of the text string and a RECENTER TEXT command is executed. If an <esc> is immediately entered, another search is initiated for the same text string. Otherwise, the search mode is exited.

BACKWARD SEARCH

- Search for ' <string> '

If the string is not found, the message "Not Found" is displayed, search mode is exited, and the cursor is left unchanged.

Not Found '<string>'

<^C> CANCEL. The CANCEL command terminates a current text search, aborts a command prompt, and interrupts an infinite macro sequence.

CANCEL

Get file ' <string> ^C'

<^D> DEFINE MACRO. The DEFINE MACRO command enters and exits macro definition mode. Macro definition mode is indicated by the presence of the word "MACRO" in the lower right hand corner of the screen. In macro mode, each entered key (except <^D> and <^E>) is stored in the MACRO BUFFER. Commands are executed as the macro is defined. The macro definition is completed by entering another <^D>.

DEFINE MACRO

MACRO

Macro definition mode is automatically terminated if the macro buffer overflows. The macro buffer holds 256 characters. Only one macro is can be defined at a time. The macro definition is deleted by entering <^D> <^D>.

(11.1.1 JED - VIRTUAL SCREEN EDITOR continued)

<^E>	EXECUTE MACRO. The EXECUTE MACRO command executes the command strings currently stored in the macro buffer. Each command is recalled and interpreted as if it had been entered from the keyboard.	EXECUTE MACRO
<^F>	FIND POINTER. The FIND POINTER command locates the editor pointer and leaves the cursor immediately to the right of the pointer, (see <^P> command). The text is recentered on the screen. If no pointer exists in the text, the message 'No pointer in text' is displayed.	FIND POINTER No pointer in text
<^G>	GET FILE. The GET FILE command, when VERIFIED, reads a file into the editor buffer. The command first clears the edit buffer, reads in the text, and places the cursor at the beginning of the buffer.	GET FILE Get file ' <string> ' VERIFY
<^H>	MOVE LEFT. The MOVE LEFT command moves the cursor left one character in the edit buffer. When used in conjunction with the <esc> key, it moves the cursor to the beginning of the line. The cursor will not move past the beginning of the line. A tab field is treated as one character.	MOVE LEFT
<^I>	TAB. Your screen is divided into 10 zones or tab stops, consisting of eight characters each. When a TAB is entered, the cursor advances to the next tab stop.	TAB
<^J>	MOVE DOWN. The MOVE DOWN command moves the cursor down one display line. When used in conjunction with the <esc> key, the cursor is moved down 11 lines. The cursor moves directly down unless 1) it moves into a tab field, or 2) it moves down past the end of the next line. In either case, the cursor then moves left to the beginning of the tab field or the end of the line. The screen scrolls up when the cursor attempts to move down past the 23rd line.	MOVE DOWN

(11.1.1 JED - VIRTUAL SCREEN EDITOR continued)

<^K>	MOVE UP. The MOVE UP command moves the cursor up one display line. When used in conjunction with the <esc> key, the cursor is moved up 11 lines. The cursor moves directly up unless 1) it moves into a tab field, or 2) it moves up past the end of the previous line. In either case, the cursor then moves left to the beginning of the tab file or the end of the line. The screen scrolls down when the cursor attempts to move up past the first line. The cursor cannot move above the beginning of the text.	MOVE UP
<^L>	MOVE RIGHT. The MOVE RIGHT command moves the cursor right one character. When used in conjunction with the <esc> key, the cursor is moved to the end of the current line. The cursor will not move past the end of the line.	MOVE RIGHT
<CR>	CARRIAGE RETURN. Each line is terminated by a <CR>.	CARRIAGE RETURN
<^N>	NEW BUFFER. The NEW BUFFER command, when VERIFIED, causes the editor to clear the edit buffer and reinitialize all flags.	NEW BUFFER New buffer VERIFY
<^O>	OUTPUT BLOCK. The OUTPUT BLOCK command, when VERIFIED, outputs all text between the cursor and the pointer to the specified file, (see <^P> command).	OUTPUT BLOCK Output cursor to Pointer VERIFY
<^P>	PLACE POINTER. The PLACE POINTER command inserts a special character called a pointer into the text buffer. This pointer is displayed as a "~". The pointer and the cursor define a segment of text which can be written to the disk <^O> or internal UP buffer <^U>, and/or deleted <^\>. If the pointer already exists in the edit buffer, the <^P> command deletes the old pointer and then inserts a new pointer at the cursor position. To get rid of it, simply delete it.	PLACE POINTER

(11.1.1 JED - VIRTUAL SCREEN EDITOR continued)

<^Q> QUIT. When VERIFIED, the QUIT command returns you to the PDOS monitor. All files associated with the editor are closed.

```
QUIT
                                        QUIT VERIFY
```

<^R> RECENTER TEXT. The RECENTER TEXT command recenters the text around the cursor in the middle of the screen (line 12).

```
RECENTER TEXT
```

<^S> SEARCH FORWARD. The SEARCH FORWARD command searches the text buffer for a specific text string. The command prompts with "Search for '", after which the text string is entered. If the string is found, the cursor is positioned to the right of the text string and a RECENTER TEXT command is executed. If an <esc> is immediately entered, another search is initiated for the same text string. Otherwise, the search mode is exited.

```
SEARCH FORWARD

        Search for ' <string> '
```

If the string is not found, the message 'Not found' is displayed and search mode is exited.

```
        Not found ' <string> '
```

<^T> TOP OF BUFFER. The TOP OF BUFFER command moves the cursor to the beginning of the edit buffer.

```
TOP OF BUFFER
```

<^U> COPY TO UP BUFFER. The COPY TO UP BUFFER command copies the text between the cursor and the pointer into an internal temporary buffer called the up buffer. If the text length is less than 255 characters, then the message 'I got it' is displayed. Otherwise, the message 'OVERFLOW' is displayed and the string is truncated in the buffer. The up buffer is inserted into the edit buffer at the cursor with the <^A> command (see ^A command). If there is not pointer in the buffer, then 'No pointer in text' is displayed.

```
COPY TO UP BUFFER

        I got it

        OVERFLOW


        No pointer in text
```

(11.1.1 JED - VIRTUAL SCREEN EDITOR continued)

<^V> CONTROL CHARACTER INSERT. The <^V> character causes the next entered character to be inserted in the edit buffer, regardless of its command definition. This allows control characters to be entered into the edit buffer. Because control characters are not displayable, they are assigned regular character representations and appear as such in the text. When searching for the character, you must remember to use the <^V> before the control character and not the displayed character.

CONTROL CHARACTER INSERT

^A = !	^I = <TAB>	^Q = 1	^Y = 9
^B = "	^J = *	^R = 2	^Z = :
^C = #	^K = +	^S = 3	^[=
^D = $	^L = ,	^T = 4	^\ = <
^E = %	^M = <CR>	^U = 5	^] = =
^F = &	^N = .	^V = 6	^^ = >
^G = '	^O = /	^W = 7	^_ = ?
^H = (	^P = 0	^X = 8	

<^W> WRITE FILE. The WRITE FILE command, when VERIFIED, writes the edit buffer to the specified file.

WRITE FILE

<^X> TYPE AHEAD CANCEL. The TYPE AHEAD CANCEL command clears the PDOS character input buffer.

TYPE AHEAD CANCEL

<^Y> INSERT FILE. The INSERT FILE command, when VERIFIED, reads and inserts the specified file into the text buffer beginning at the cursor. The cursor is placed at the beginning of the inserted text.

INSERT FILE

<^Z> BOTTOM OF BUFFER. The BOTTOM OF BUFFER command moves the cursor to the end of the edit buffer.

BOTTOM OF BUFFER

<^\> DELETE BLOCK. The DELETE BLOCK command, when VERIFIED, deletes the segment of text between the cursor and the pointer, including the pointer.

DELETE BLOCK

<^]> CLEAR TO END OF LINE. The CLEAR TO END OF LINE command deletes the text from the cursor to the end of the line.

CLEAR TO END OF LINE

<^^> DELETE LINE. The DELETE LINE command deletes the text from the cursor to the end of the line, including the carriage return.

DELETE LINE

(11.1.1 JED - VIRTUAL SCREEN EDITOR continued)

<^_> DELETE RIGHT. The DELETE RIGHT command deletes the character to the right of the cursor.

DELETE RIGHT

<rubout> DELETE LEFT. The DELETE LEFT command deletes the character to the left of the cursor.

DELETE LEFT

ESCAPE FUNCTIONS

The escape key has two functions. First, the <esc> is used to end the file name or search string parameter for the <^G>, <^O>, <^W>, and <^S> commands. The string is then terminated with a single quote and the editor continues.

<esc> terminates parameter string

The second function of the <esc> key involves JED modes and PDOS access. The single character cursor move commands (<^H>, <^L>, <^J>, <^K>) can be changed to multiple character moves by entering the <esc> key before the command control key. The editor remains in this mode until a key other than <^J> or <^K> is entered.

<esc> selects JED modes

The REPLACE and INSERT modes are selected with the two character commands <esc> <^R> and <esc> <^I>. A PDOS directory listing is displayed by entering <esc> <^A>. These and other <esc> commands are listed below. Any character following an <esc>, other than those listed, cause the <esc> to be ignored.

<esc> <^A> LIST DIRECTORY. The LIST DIRECTORY command allows you to examine the disk directory contents without exiting the editor. The prompt, "List directory'", is issued, and you input a list parameter string, just as with the 'LS' command of PDOS, and close it with an <esc>. The specified file directory is scrolled to the screen. When the list is completed, the message, 'Strike any key :', is displayed. The next key struck is not processed by JED, but the screen is cleared and then refreshed to display the edit buffer again.

LIST DIRECTORY

List directory'

Strike any key :

(11.1.1 JED - VIRTUAL SCREEN EDITOR continued)

‹esc› ‹^E›	INFINITE MACRO. The INFINITE MACRO mode will repeat the editor macro until either an error occurs or a ‹^C› is entered.	INFINITE MACRO
‹esc› ‹^H›	JUMP LEFT. The JUMP LEFT command moves the cursor to the beginning of the line. (Left arrow)	JUMP LEFT
‹esc› ‹^I›	INSERT MODE SELECT. The INSERT MODE SELECT command causes characters to be inserted into the text string while all characters to the right and down are automatically adjusted. (This is the default mode.)	INSERT MODE SELECT
‹esc› ‹^J›	JUMP DOWN. The JUMP DOWN command moves the cursor down eleven lines. This is used to scan the text quickly. (Down arrow)	JUMP DOWN
‹esc› ‹^K›	JUMP UP. The JUMP UP command moves the cursor up eleven lines. (Up arrow)	JUMP UP
‹esc› ‹^L›	JUMP RIGHT. The JUMP RIGHT command moves the cursor to the end of the line. (Right arrow)	JUMP RIGHT
‹esc› ‹^R›	REPLACE MODE SELECT. The REPLACE MODE SELECT command allows characters under the cursor to be overwritten instead of moving the old characters to the right and inserting the new ones.	REPLACE MODE SELECT

11.1.2 JEDY - NONVIRTUAL EDITOR

JEDY is a non-virtual memory implementation of the JED editor. All features of JED are retained, with the exception that editing must be done completely within user memory. When the internal memory limits are exceeded, a 'BUFFER FULL' message is printed by JEDY.

JEDY is advantageous where disks are often swapped in and out. Also, a larger buffer can be accommodated since the virtual memory handlers have been omitted.

Additions to the JEDY editor include:

<esc> <^B> FREE BYTE COUNT. The FREE BYTE COUNT command reports to the console the number of bytes remaining in the buffer. When this count becomes zero and you enter another character, JEDY reports a 'BUFFER FULL' error, rings the terminal bell, and ignores the character input. The only commands accepted from then on are movement and deletion commands.

```
FREE BYTE COUNT

        Free bytes =
```

11.1.3 EDIT - CHARACTER EDITOR

The EDIT program is a character oriented editor. Single or double character commands, optionally preceded by a number, allow you to edit a text file in memory. EDIT can be exited and re-entered without destroying the buffer.

A character buffer is used for string editing. An imaginary pointer indicates where in the buffer all editing takes place. This pointer is easily moved around without disturbing the buffer data.

A file can be read in by EDIT when it is run from PDOS by simply following EDIT with the desired file name. PDOS loads the EDIT utility and then EDIT opens the file for input (GI<filename>), yanks in a page (Y), and closes the file (GI).

```
.EDIT FILE1
*GIFILE1$Y$GI$
*
```

EDIT prompts for all commands with a '*' character. The first character of each string entered is the command byte. Commands are terminated with the <esc> key which echoes as a '$'. Most commands have two parts in which case the <esc> is also used to delimit the arguments. Commands are not executed unit a double <esc> is entered. Commands can be chained together as long as the double <esc> is not entered. Once a double <esc> is entered, execution begins. The execution of the command line can be interrupted by a <^C>.

EDIT file storage is page oriented. Data is read from an input file until either a form feed <^L> or end-of-file is found. An output file then receives the edited data. The normal output commands again place the form feed between pages. However, this can be overridden for special cases.

(11.1.3 EDIT - CHARACTER EDITOR continued)

The EDIT commands are defined as follows:

A	APPEND NEXT PAGE. The next page from the input file is appended to the end of the edit buffer. A form feed <^L> is not inserted. The pointer is placed at the beginning of the new page.	APPEND NEXT PAGE
B	MOVE TO BEGINNING OF TEXT. The pointer is placed at the beginning of the edit buffer.	MOVE TO BEGINNING OF TEXT
C<s1>$<s2>	CHANGE. A forward search is done for <s1> from the current buffer pointer. If found, <s1> is replaced by <s2>. The pointer is placed at the beginning of the changed string. If <s1> is not found, the pointer is left unchanged.	CHANGE
#D	CHARACTER DELETE. # characters are deleted from the edit buffer, beginning at the pointer.	CHARACTER DELETE
GI<s>	GET FOR INPUT. The PDOS file specified by <s> is opened for input. If a previous file was already open, it is first closed.	GET FOR INPUT
GI	CLOSE INPUT FILE. If a file is currently open for input, it is closed.	CLOSE INPUT FILE
GO<s>	GET FOR OUTPUT. The PDOS file specified by <s> is opened for output. If a previous file was already open, it is first closed.	GET FOR OUTPUT
GO	CLOSE OUTPUT FILE. If a file is currently open for output, it is closed.	CLOSE OUTPUT FILE
H	RETURN TO PDOS. EDIT exits to the PDOS monitor.	RETURN TO PDOS
#I	INSERT BYTE. Control characters (including a <^C> and <esc>) can be inserted into the edit buffer by preceding the insert command with the decimal equivalent of the character.	INSERT BYTE

(11.1.3 EDIT - CHARACTER EDITOR continued)

I<s>	INSERT STRING. The string <s> is inserted into the edit buffer beginning at the pointer. The string <s> is terminated by the <esc> character.	INSERT STRING
#J	MOVE LINES. The HERE POINTER ('.') sets another pointer into the edit buffer. Lines can be moved from the HERE pointer to the current buffer pointer by the MOVE LINES command. # indicates how many lines are to be moved.	MOVE LINES
#K	KILL LINES. # lines, delimited by a <CR>, are deleted from the edit buffer. The buffer pointer is the point of transaction.	KILL LINES
#L	LINE MOVE. The buffer pointer is moved # lines forward or backward from its current position. A zero (or no parameter) move is to the beginning of the current line.	LINE MOVE
#M	CHARACTER MOVE. The buffer pointer is moved # characters forward or backward from its current position.	CHARACTER MOVE
N	NEW BUFFER. When this command is verified, the edit buffer is cleared.	NEW BUFFER
P	PUNCH BUFFER. The entire edit buffer is written to the output file and a form feed <^L> is appended to the end.	PUNCH BUFFER
#P	PUNCH # LINES. # lines, beginning at the pointer, are written to the output file. A form feed <^L> is appended to the end.	PUNCH # LINES
PW	PUNCH WITHOUT <^L>. The entire edit buffer is written to the output file.	PUNCH WITHOUT <^L>
#PW	PUNCH # LINES WITHOUT <^L>. # lines, beginning at the pointer, are written to the output file.	PUNCH # LINES WITHOUT <^L>

(11.1.3 EDIT - CHARACTER EDITOR continued)

S<s>	SEARCH FOR STRING. A search for string <s> is made in the edit buffer beginning at the pointer. If <s> is found, the pointer is placed at the beginning of the string. If <s> is not found, the pointer is left untouched.	SEARCH FOR STRING
T	TYPE ENTIRE BUFFER. The entire buffer is output to the user console. A ^C will stop the printing.	TYPE ENTIRE BUFFER
#T	TYPE # LINES. # lines of edit text, beginning at the pointer, are printed to the user console. If # is negative, then the pointer is moved back # lines and # lines are printed.	TYPE # LINES
#X	EXECUTE MACRO. The edit macro, if defined, is executed # times. If no number is given, the macro will be executed only once. A <^C> breaks execution.	EXECUTE MACRO
XM<s>	DEFINE MACRO. An edit macro is defined by the string <s>. No execution takes place.	DEFINE MACRO
XM	DELETE MACRO. The edit macro is deleted.	DELETE MACRO
Y	YANK NEW PAGE. The edit buffer is cleared and a new page of characters is read from the input file. This read is delimited by a form feed <^L> or end-of-file.	YANK NEW PAGE
Z	MOVE TO END OF TEXT. The pointer is moved to the end of the edit buffer.	MOVE TO END OF TEXT
.	HERE POINTER. A pointer is set for the MOVE LINES command. The current line number is also printed to the user console.	HERE POINTER
:	# OF LINES. The number of lines in the user edit buffer is printed to the user console.	# OF LINES
=	# OF CHARACTERS. The number of characters in the user edit buffer is printed to the user console.	# OF CHARACTERS

(11.1.3 EDIT - CHARACTER EDITOR continued)

+	# OF CHARACTERS LEFT. The number of characters available in the edit buffer is printed to the user console.	# OF CHARACTERS LEFT

A sample edit session is given below:

```
.EDIT
EDIT R2.4
*I* THIS IS A TEXT FILE
*
START   XPMC            ;OUTPUT MESSAGE
*         DATA MESO1
        XEXT
*
MESO1   BYTE >OA,>OD
        TEXT 'OUTPUT MESSAGE'
        BYTE 0
        END START
$$
*T$$
* THIS IS A TEXT FILE
*
START   XPMC            ;OUTPUT MESSAGE
          DATA MESO1
        XEXT
*
MESO1   BYTE >OA,>OD
        TEXT 'OUTPUT MESSAGE'
        BYTE 0
        END START

*B$$
*1T$$
* THIS IS A TEXT FILE

*SOUTPUT$L$1T$$
START   XPMC            ;OUTPUT MESSAGE

*C'OUTPUT$'PRINTED$L$1T$$
        TEXT 'PRINTED MESSAGE'

*GOTEMP$$
*P$GO$$
*H$$
.
```

11.2 ASM - PDOS 9900 ASSEMBLER

ASM is a TMS 9900 assembler designed to be used with the PDOS operating system. ASM accepts TMS 9900 and 9995 assembly mnemonics and directives, and outputs 9900 tag object code. It can be run in foreground mode with the user inputting optional files from the keyboard, or in background mode as an offline task while other processes such as editing are run in the foreground.

Input and output options are specified by a list of file names following the ASM command or from keyboard prompts. These options are, in order:

<SOURCE>	Assembly source file (required)
<OBJECT>	TI Object output file
<LIST>	Assembly listing file
<ERROR>	Assembly error file (default to console)
<XREF>	Symbol cross reference file

```
.ASM TEST:SR,TEST,LIST,,XREF
ASM R2.4
SOURCE=TEST:SR
OBJ=TEST
LIST=LIST
ERR=
XREF=XREF
END OF PASS 1
0 DIAGNOSTICS
END OF PASS 2
0 DIAGNOSTICS
```

11.2.1 USING THE ASSEMBLER

To invoke the assembler from the keyboard, simply insert a disk with an ASM file on it and type ASM. If there are any disk errors encountered, such as 'file not defined', then ASM will report 'PDOS ERROR = ', followed by the PDOS error #, and then return to PDOS. Since some programs create LIST or XREF files too large for the disk space available, these files can be specified as drivers for printers or terminals.

```
.ASM
ASM R2.4
SOURCE=_
ASM DEMO:SR,DEMO,$TTA,,$TTO
```

The SOURCE file consists of TI9900 assembler directives and mnemonics as described in the TI9900 assembler manual. If there is no IDT directive, no IDT will be output. The SOURCE file must end with either a LINK or END directive. The LINK command opens the file specified in the operand field and continues assembling as if it were appended to the SOURCE file. The final file must end with an END directive, which causes the assembler to reopen the original SOURCE file and begin the second pass. The argument of the END directive is an expression whose value is output to the OBJECT file with an entry tag.

```
                                SOURCE FILE


BEGIN   MOV R0,R0
        ...
LINK PROG2

END BEGIN
```

(11.2.1 USING THE ASSEMBLER continued)

OBJECT FILE

TI9900 tag object code is output to the file selected by the 'OBJ=' prompt. The OBJECT code is output on the second pass as a string of ASCII characters. The format is defined by the TI9900 assembly manual.

ASM and PDOS supports byte addresses. PDOS loads two bytes at a time on byte boundaries. Checksums are optional in the OBJECT file. All records are terminated with the 'F' tag.

The assembly symbol table can be optionally dumped to the OBJECT file by selecting the 'S=1' option of the 'OPT' directive. These values could be used by a symbolic debugger at a later time.

The OBJECT file is closed by the assembler with an OB attribute. If no linking is required, PDOS can execute the file directly.

LIST FILE

A listing of the source code, along with the assembled object code values, is generated when a file name is entered for the LIST option. The LIST file is a paged with the assembler name and revision at the top. The page number, date, SOURCE file name, and disk name are on the next line.

A source line is preceded by a two digit line number, the hexadecimal address, and up to three hexadecimal object code values. The assembler automatically pages every 56 lines or when the 'PAGE' directive is encountered.

Following the source listing, an alphabetized list of the symbols defined during assembly, along with the symbol type and value, is output to the LIST file. Possible symbol types are:

A = absolute
R = program-relocatable
D = data-relocatable
U = undefined
M = multiply-defined
E = REF symbol

The symbol table can be optionally replaced with a symbol cross reference by selecting the 'X=1' option of the 'OPT' directive.

(11.2.1 USING THE ASSEMBLER continued)

The assembler reports any assembly errors by printing the line from the listing to the ERROR file, with an error letter in column 3. Only the first error on each line is reported by a letter, but the "total errors" message counts all errors found by the assembler. If no error file is specified, errors are printed to the console. The current errors are defined as:

'B' = Byte overflow
'C' = Illegal ASCII constant
'D' = Text delimiter error
'E' = Illegal number
'F' = File error
'M' = Multiply defined symbol
'N' = Numeric overflow
'O' = Field overflow
'R' = CRU displacement out of range
'S' = Illegal symbol
'T' = Symbol table overflow
'U' = Undefined symbol
'X' = Missing operand
'e' = Expression mode error
'f' = Floating point conversion error
'm' = Multiply-defined symbol referenced
't' = Truncation error

ERROR FILE

```
16X0006: C000          MOV R0,
22A0018: 1000          JMP $+>220
```

A cross reference is output to the specified XREF option file in a paged form, similar to the LIST file. Each symbol is listed with its type and value followed by the page and line # of each reference in the source. The reference at which the symbol is defined is marked with an asterisk (*). The workspace registers R0 through R15 are also included in the cross reference. The cross reference is done during the second pass, eliminating the need for a third pass. The XREF listing is paging is consecutive with that of the LIST file.

XREF FILE

```
C       A 000D  1/11*   1/29
D       M 00A9  1/4*    1/9*    1/29
```

It is possible to direct the cross reference of the file directly to the LIST file, by setting the X option flag nonzero with the OPT directive. This eliminates the need to append the XREF file to the LIST file for printing.

11.2.2 ASSEMBLY LANGUAGE FORMAT

Assembly language source statements consist of the following four fields:

```
    LABEL  MNEMONIC  OPERANDS  COMMENT
```

```
LABEL   A R1,R2         ;ADD
```

The source line must be less than 108 characters long, and at least one blank or TAB must be inserted between fields.

LABEL FIELD

The label is a symbol consisting one to six characters, beginning with an alphabetic character in position one of the source line. The label field is terminated with at least one blank or TAB character. If a label is not used, character position one must be a blank or tab character.

MNEMONIC OR OPCODE FIELD

This field contains the mnemonic code of 1) a 9900 instruction, 2) an assembler directive, 3) a symbol representing one of the program defined XOPs, or 4) a special code invoking a PDOS command primitive. Usually this field is positioned in the second tab field, beginning eight characters from the left. All of the four character PDOS command primitives are legal opcodes.

```
START   XGNP            ;GET PARAMETER
          JNE ERR       ;NONE
        ...
```

OPERAND FIELDS

The operands specify the memory locations or immediate data to be used by the instruction. Constants, symbols, literals, and expressions are legal operands.

COMMENT FIELD

Comments follow the operand field. Usually the comment field is positioned in the fourth tab field, beginning 24 characters from the left. The use of a semicolon as the first character in the comment field helps to set off comments for clarity. If the first character of a source line is an asterisk (*), then the entire line is a comment.

11.2.3 CONSTANTS

Constants can be signed decimal, hexadecimal, or binary integers, ASCII constants, or 6-byte floating point numbers.

Decimal integers are written as a string of numerals in the range of -32768 to +32767.

```
AI R5,20
```

Hexadecimal constants consist of a string of hexadecimal digits preceded by a right angle bracket and range from 0 to >FFFF.

```
ORI R3,>FF00
```

Binary constants consist of a string of 1's and 0's, preceded by a percent sign (%).

```
LI R0,%10110110
```

ASCII character constants are one or two characters enclosed in single quotes. A single quote can be entered by using two quotes ('').

```
LI R0,'AB'
LI R0,'''B'
```

Floating point constants use the 'CONS' directive and may be any legal floating point number including scientific notation using the 'E' operator.

```
CONS 3.1415926,1E10
```

11.2.4 SYMBOLS

Symbols begin with an alphabetic character and can be up to six characters in length. There can be no embedded blanks. Legal characters for positions 2 through 6 are A-Z, 0-9, ., % and $. The assembler predefines the dollar sign ($) to represent the current location counter, and the symbols R0 through R15 are used to represent the workspace registers.

```
DATA LABEL1,LABEL2,A$,L0%
```

A given symbol can be used as a label only once, and any symbol in the OPERANDS field must have been used as a label previously. Symbols defined with the DXOP directive are used in the OPCODE field.

11.2.5 OPERATORS

The binary operators interpreted by ASM for expressions are +, -, *, /, &, !, and \. These operators perform 16-bit addition, subtraction, signed multiplication, signed division, logical AND, logical inclusive OR, and logical shift right, respectively. To shift an operand logically left, use a negative shift count.

```
DATA A+B-C*D/E
DATA -1&-2!FLAG
BYTE ADDR&>00FF*256

BYTE ADDR\8,ADDR\-8
```

11.2.6 EXPRESSIONS

Expressions are made of up symbols and constants, which may be immediately preceded by a unary plus (+) or minus (-). Symbols and constants are separated by operators, and the expression is evaluated from left to right with no operator precedence. Only absolute operands are used with multiplication or division. The rule governing addition and subtraction of relocatable operands is described in the TI assembly manual.

```
LI R0,ASM42-ASM30/2+1300+100
MOVB @-PEND(R5),@TABLE+INDEX0
```

11.2.7 ASSEMBLER OBJECT TAGS

ASM outputs standard TI 9900 tag object and record formats, with the following exceptions:

1) Any tag address can be on an odd byte boundary.

2) External tags 3, 4, 5, and 6 are modified to be used with the PDOS single pass linker.

3) Tags G and H have been added for DSEG addresses and data.

4) Tags 7, 8, D, and E are not part of ASM output.

The data type is reflected in the listing by a single character which follows the hex object list. These character types and the assembler output tag definitions are defined in Table 11.1.

A SYSTEM FILE (type=SY) is a modified form of 9900 tag object code. A one-third reduction in object code results from: 1) eliminating all data superfluous to PDOS, including redundant address codes, checksums, <line feeds>, and IDTs, and 2) converting all necessary data from four ASCII characters into two binary bytes. A SYSTEM FILE is generated from an object file by the SYFILE utility. Only tags 2, A, B, and C are output in the system file.

```
Type = SY
```

```
Tags 2,A,B,C
```

The PDOS loader will accept either 9900 tag object or a system object file. Since system object files are smaller, load time is proportionately faster.

(See 5.30 LOAD FILE and 12.23 SYFILE.)

TAG	FIELD 1	FIELD 2	FUNCTION	LIST ID
0	PSEG length	Program ID (8)	IDT	
1	Absolute entry	(not used)	END absolute	
2	Relocatable entry	(not used)	END relocatable	
3	Symbol (6)	(not used)	External REF	+
4	Absolute value	Symbol (6)	External DEF	
5	P-R value	Symbol (6)	P-R External DEF	
6	D-R value	Symbol (6)	D-R External DEF	
7	(Reserved)			
8	(Reserved)			
9	Absolute Address	(not used)	AORG	
A	P-R Address	(not used)	RORG,PSEG	
B	Absolute Data	(not used)	Absolute	blank
C	P-R Data	(not used)	Program-relocatable	'
D	(Reserved)			
E	(Reserved)			
F	(not used)	(not used)	End of record	
G	D-R Address	(not used)	RORG,DSEG	
H	D-R Data	(not used)	Data-relocatable	"

TABLE 11.1 ASSEMBLER OBJECT TAGS

11.2.8 ASSEMBLER DIRECTIVES

The PDOS ASM assembler supports the following directives:

```
IDT ('name')            Program identifier
END <exp>               End assembly, set entry
LINK <file>             Link to file

EQU <exp>               Define assembly-time constant
BYTE <exp1>{,<exp2>}... Initialize byte
DATA <exp1>{,<exp2>}... Initialize word
TEXT {-}{+}..('string') Initialize text (any delimiter)
CONS <fp #>{,<fp #>}... Initialize 6 byte FP number

AORG <exp>              Absolute origin
RORG {<exp>}            Relocatable origin
DORG <exp>              Dummy origin (no object)
PSEG                    Program segment
DSEG                    Data segment

BES <exp>               Block ending with symbol
BSS <exp>               Block starting with symbol
EVEN                    Set word boundary

PAGE                    Page eject
TITL ('string')         Page title
LIST                    List source
UNL                     No source list
DXOP <symbol>,<exp>     Define extended operation

DEF <sym>{,<sym>}...    External definition
REF <sym>{,<sym>}...    External reference
COPY <file>             Include from <file>

IFZ <exp>,<symbol>      If <exp> zero, goto <symbol>
IFN <exp>,<symbol>      If <exp> nonzero, goto <symbol>
DUP <exp>               Duplicate next line <exp> times
OPT <char>=<exp>{,<char>=<exp>}...
                        Set option flag <char>
        ?       QFLG    Assemble if ? nonzero
        #       PFLG    Assemble right or left half
        C       CFLG    Output tag 7 checksums with object
        L       LFLG    Expanded list options
        R       RFLG    Register cross reference
        S       SFLG    Punch symbol table to object
        X       XFLG    Output XREF to LIST file
```

11.2.8.1 REQUIRED DIRECTIVES

Each source program must contain an IDT, and END or LINK. These directives control the flow of input to the assembler.

END - PROGRAM END

Syntax definition:

{<label>} END {<exp>} {<comment>}

The END directive terminates the assembly source file. Any source statements following the END directive are ignored by the assembler. When the label field is used, the current value of the location counter is assigned to the symbol. The operand field is optional. It specifies a program-relocatable or absolute entry point to the program.

```
BEGIN   ...
        END BEGIN       ;OUT ENTRY TAG

MODUL   ...
        END             ;NO ENTRY TAG
```

If the assembler finds an external reference or a data-relocatable expression in the operand field, an expression mode error (e) is printed, and no entry point is output to the object file. The comment field may only be used when the operand field is present.

An entry point in the OBJECT file consists of a '1' tag followed by an absolute entry address or a '2' tag followed by a relocatable address. PDOS requires a relocatable entry address to execute the file from the monitor. When a program is to be combined with other modules by the LINKer utility, the entry address is optional.

NOTE: The source line containing the END directive MUST have a carriage return <CR> at the end, or a PDOS error 56 will be printed and the assembler will abort at the end of the first pass.

(11.2.8.1 REQUIRED DIRECTIVES continued)

IDT - PROGRAM IDENTIFIER

Syntax definition:

{<label>} IDT <'name'> {<comment>}

IDT assigns a name to a program. An IDT directive must precede any code that results in object code. When the label field is used, the current value in the location counter is assigned to the label. The operation field contains IDT. The operand field contains the program name <'name'>, a character string of up to eight characters, delimited by any character.

If the IDT has no operand, the assembler outputs a missing operand error (X) and ignores the line. When an operand of more than eight characters is entered, the assembler prints a truncation error (t) and outputs in the IDT field, the first eight characters. If less than eight characters are entered, the assembler fills the IDT field with blanks. If no closing delimiter is encountered, the assembler prints a delimiter error (D) and outputs the IDT.

The program name is placed in the object code for utilities such as ALOAD and LINKer, but serves no purpose during the assembly. The assembler outputs to the OBJECT file a 'O' tag, followed by 4 hexadecimal digits representing the PSEG length, and the 8 character IDT string.

```
        IDT 'PRGM1'
        IDT     $PRGMR2.O$
```

LINK - LINK TO FILE

Syntax definition:

{<label>} LINK <file> {<comment>}

The LINK directive closes the current source file, opens the file specified by the operand, and continues the assembly process. The last statement of a source file must be either the END or LINK directive. Any source statements following a LINK directive are ignored and the assembler begins reading source statements from the specified <file>.

When the label field is used, the current value of the location counter is assigned to the symbol. The operation field contains LINK. The operand field contains the name of a PDOS file, from which the assembler reads subsequent source statements. The comment field is optional.

```
*       FILE ASM1:SR
ASM     ...
        LINK ASM2:SR

*       FILE ASM2:SR
        ...
        LINK ASM3:SR

*       FILE ASM3:SR
        ...
        END ASM
```

11.2.8.2 CONSTANT INITIALIZATION DIRECTIVES

The following directives are used to initialize constants, labels, data, and text in source programs. They are EQU, BYTE, DATA, CONS, and TEXT.

EQU - DEFINE CONSTANT

Syntax definition:

```
<label>        EQU <exp>        {<comment>}
```

The EQU directive assigns a value to a symbol. The label field contains the symbol and the operand field contains the value. Use of the comment field is optional. The value must be consistent for each pass of the assembler.

```
FLAG    EQU 1
REGO    EQU 'R0'
EMUL    EQU FLAG&1*256
```

BYTE - INITIALIZE BYTE

Syntax definition:

```
{<label>}        BYTE <exp1>{,<exp2>}... {<comment>}
```

The BYTE directive defines values for one or more successive bytes of memory. When the label field is used, the location at which the assembler places the first byte is assigned to the label.

The operand field contains one or more expressions separated by commas. There can be no embedded blanks nor external references. The assembler evaluates each expression and places the value at the current memory location as an 8-bit two's complement number. The memory address is incremented by one. When truncation is required, the assembler prints a byte overflow error (B) and places the rightmost portion of the value in the byte. The comment field is optional.

```
        BYTE >07,>0A,>0D,0
        BYTE 'A','B',0
```

(11.2.8.2 CONSTANT INITIALIZATION DIRECTIVES continued)

DATA - INITIALIZE WORD

Syntax definition:

{<label>} DATA <exp1>{,<exp2>}... {<comment>}

The DATA directive defines values for one or more successive words of memory. The assembler first advances the location counter to a word (even) boundary. When the label field is used, the location at which the assembler places the first word is assigned to the label.

```
DATA 0,1,2,'AB','*'*256
```

The operand field contains one or more expressions separated by commas. There can be no embedded blanks, but external references are allowed if no arithmetic is performed on them. The assembler evaluates each expression and places the value in a word as a sixteen-bit two's complement number. The memory addresses is incremented by two. When truncation is required, the assembler prints a numeric overflow error (N) and places the rightmost portion of the value in the word. The comment field is optional.

CONS - INITIALIZE 6 BYTE FLOATING POINT NUMBER

Syntax definition:

{<label>} CONS <fp #>{,<fp #>}... {<comment>}

The CONS directive defines floating point values for one or more successive 6-byte memory locations. The assembler first advances the location counter to a word (even) boundary. When the label field is used, the location at which the assembler places the first word is assigned to the label.

```
CONS 3.1459,1.234E-10,.0000001
```

The operand field contains one or more floating point expressions separated by commas. There can be no embedded blanks. The assembler converts each floating point number into the standard PDOS 6-byte IBM excess 64 format and places the value in three consecutive words. The memory address is incremented by six. When a conversion error occurs, the assembler prints a floating point conversion error (f) and places zeros in the three words. The comment field is optional.

(11.2.8.2 CONSTANT INITIALIZATION DIRECTIVES continued)

TEXT - INITIALIZE TEXT

Syntax definition:

{<label>} TEXT {-}{+}..<'string'> {<comment>}

The TEXT directive places one or more characters in successive bytes of memory. The assembler can optionally negate the last character of the string or append a null byte to the end of the string. When the label field is used, the location at which the assembler places the first character is assigned to the label.

```
MES1    TEXT 'START PROCESS'
        BYTE >0A,>0D,0

MES2    TEXT - $NEGATE TH' LAST BYTE$

MES3    TEXT  +"TERMINATE WITH NULL BYTE"
```

The operand field contains a string delimited by any printable character except a blank, minus, or plus sign. If the comment field is not used, the closing delimiter is optional. If a blank or control character is used as a delimiter, the assembler will print a text delimiter error (D) and the line will be ignored.

Each minus sign preceding the string decrements a counter, which is initialized to zero. Each plus sign preceding the string increments the same counter. The sign of the resulting counter determines how to process the string. If the resultant counter equals zero, no action is taken. If the counter is negative (more minus signs than plus signs), then the last byte of the string is negated. If the counter is positive (more plus signs than minus signs), then an extra null byte is output after the end of the string. This has the same effect as if the TEXT statement line were followed by a BYTE 0 directive.

NOTE: The length of the string must not be so large that the entire source line exceeds 108 characters. No error message will be output if the assembler truncates a source line upon input.

11.2.8.3 LOCATION COUNTER DIRECTIVES

The following directives are used to alter the location counter during assembly, and to perform all necessary mode control, segment definition, and block assignment. They are AORG, RORG, DORG, PSEG, DSEG, BES, BSS, and EVEN.

The directives AORG and RORG control the origin modes for code generation. Directives PSEG and DSEG select program relocatable and data relocatable location counters. The directive DORG disables object output.

These directives allow you to develop applications for RAM and EPROM systems using the same source files. This is accomplished by specifying in the source a program segment (for EPROM) and a data segment (for RAM). The LINKer utility then adjusts the tagged object for PDOS or for burning into EPROM for application testing.

The complete interaction of these directives as implemented in the assembler is defined in Table 11.2 which follows DSEG.

AORG - ABSOLUTE ORIGIN

Syntax definition:

{<label>} AORG <exp> {<comment>}

The AORG directive places a value in the location counter and defines the succeeding locations as absolute. Object output is enabled. When the label field is used, it is assigned the value that the directive places in the location counter.

```
AORG >100
...
```

The operand field contains a well defined, absolute expression. If the expression is not absolute, the assembler prints an expression mode error (e) and the line is ignored. Use of the comment field is optional.

(11.2.8.3 LOCATION COUNTER DIRECTIVES continued)

RORG - RELOCATABLE ORIGIN

Syntax definition:

{<label>} RORG {<exp>} {<comment>}

The RORG directive places a value in the location counter. If encountered in absolute code, it also defines succeeding locations as program relocatable. The RORG directive enables output to the object file. When the label field is used, it is assigned the value that the directive places in the location counter. The operand field is optional. The comment field may be used only when the operand field is used.

```
RORG $
...
RORG $+>200
```

If the RORG directive appears in absolute or program relocatable code and the operand field is not used, the location counter is set to the current length of the program segment (PLEN) and the mode of the data that follows is program relocatable (P-REL).

If the RORG directive appears in data relocatable code without an operand, the location counter is set to the length of the data segment (DLEN) and the mode of the data that follows remains data relocatable (D-REL). When the operand field is used, the operand must be an absolute, program relocatable, or data relocatable expression. The expression can contains only previously defined symbols.

If the RORG directive is encountered in absolute code, a relocatable operand must be program relocatable (P-REL). If the RORG directive is encountered in relocatable code, the relocation type must match that of the current location counter. Otherwise, the assembler prints an expression mode error (e) and the line is ignored.

When the RORG directive appears in absolute code, it changes the location counter mode to program relocatable (P-REL) and replaces its value with the operand value. In relocatable code, the operand value replaces the current location counter value, and the mode of the location counter remains unchanged.

Please verify with Table 10.2.

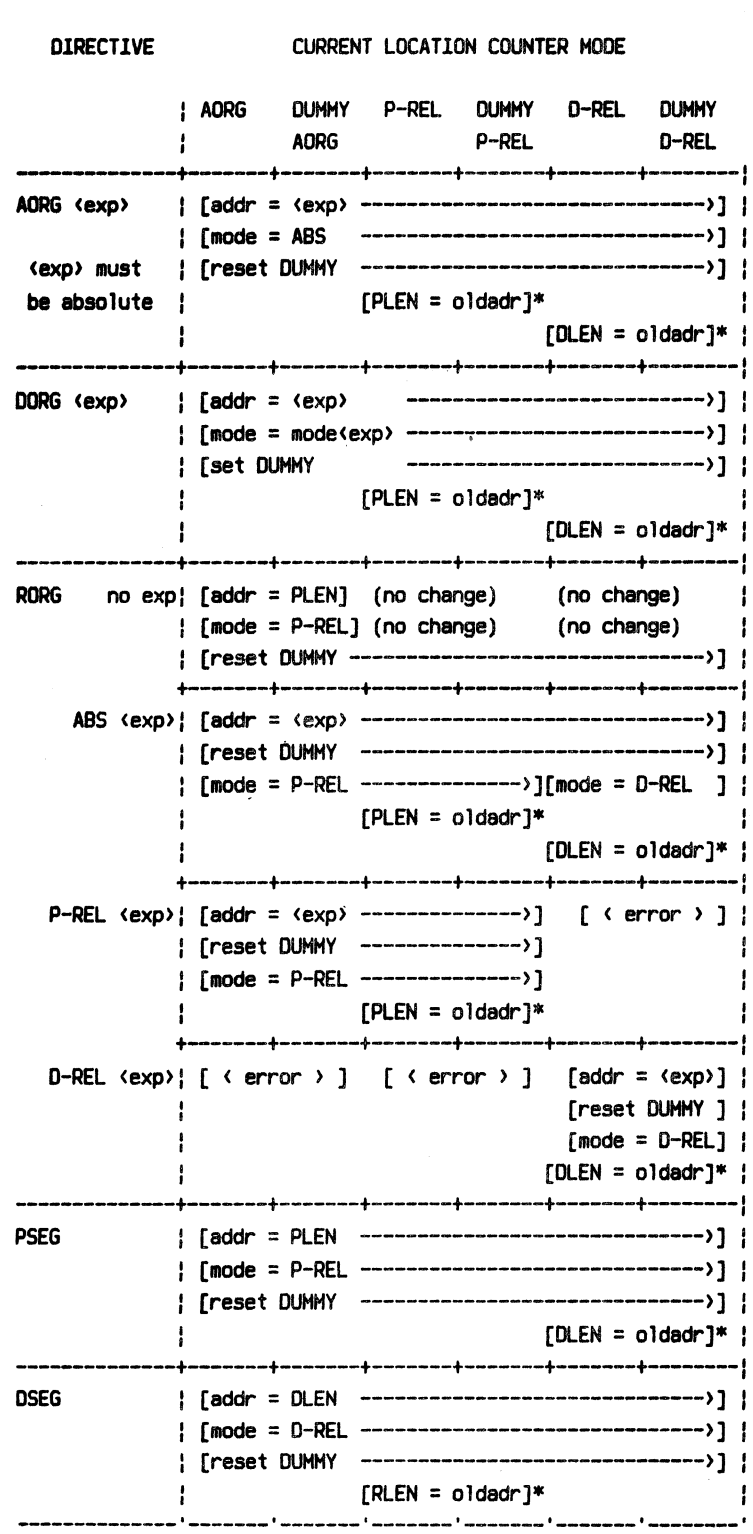


TABLE 11.2 LOCATION COUNTER DIRECTIVES

(11.2.8.3 LOCATION COUNTER DIRECTIVES continued)

DORG - DUMMY ORIGIN

Syntax definition:

{<label>} DORG <exp> {<comment>}

The DORG directive places a value in the location counter and defines the succeeding locations as a dummy block. The assembler does not generate object code in dummy mode, but operates normally in all other respects. When the label field is used, the label is assigned the value that the directive places in the location counter.

```
RORG $
...
DORG $          ;TURN OFF OBJECT
...
RORG $          ;TURN ON OBJECT
...
```

The operand field contains either an absolute or relocatable expression. Any symbol in the expression must be previously defined. If not, the assembler prints an expression mode error (e) and the line is ignored. The value of the expression replaces the location counter and the mode of the expression becomes the mode of succeeding locations.

Any of the following directives reset the dummy flag and enable output to the object code file: AORG, RORG, PSEG, and DSEG. An example of the use of the DORG directive is alternating a 'RORG $' command with a 'DORG $' command. This would cause the assembler to assign successive locations to the source code, but turn on and off the output to the object file.

PSEG - PROGRAM SEGMENT

Syntax definition:

{<label>} PSEG {<comment>}

The PSEG directive places a value in the location counter and defines succeeding locations as program relocatable. When a label is used, it is assigned the value that the directive places in the location counter. The operand field is not used and the comment field is optional.

```
RORG 0
...
DSEG
...
PSEG
...
```

The location counter is loaded with the program relocatable segment length, PLEN. The assembler sets PLEN to zero at the beginning of each pass, and updates it to the next available location whenever the program relative mode (P-REL) is exited.

(11.2.8.3 LOCATION COUNTER DIRECTIVES continued)

The assembler begins each assembly in program relocatable mode. P-REL mode is terminated with any of the following directives:

AORG
DORG where mode of <exp> is not P-REL
DSEG

PLEN is updated to the maximum value previously attained by the location counter as a result of the assembly of any preceding block of program-relocatable code. The governing equation for PLEN when exiting the P-REL mode is:

PLEN = maximum (PLEN,oldaddr).

DSEG - DATA SEGMENT

Syntax definition:

{<label>} DSEG {<comment>}

The DSEG directive places a value in the location counter and defines succeeding locations as data relocatable. When a label is used, it is assigned the value that the directive places in the location counter. The operand field is not used and the comment field is optional.

The location counter is loaded with the data relocatable segment length, DLEN. The assembler sets DLEN to zero at the beginning of each pass, and updates it to the next available location whenever the data relocatable mode (D-REL) is exited. The assembler begins each assembly in program relocatable mode, and the only way to enter data relocatable mode is through the DSEG directive. D-REL mode is terminated with any of the following directives:

AORG
DORG where mode of <exp> is not D-REL
PSEG

DLEN is updated to the maximum value previously attained by the location counter as a result of the assembly of any preceding block of data relocatable code. The governing equation for DLEN when exiting the D-REL mode is:

DLEN = maximum (DLEN,oldaddr).

(11.2.8.3 LOCATION COUNTER DIRECTIVES continued)

BES - BLOCK ENDING WITH SYMBOL

Syntax definition:

{<label>} BES <exp> {<comment>}

The BES directive advances the location counter according to the value in the operand field. The label field symbol is assigned the new location counter value.

The operand field contains a well defined, absolute expression that represents the number of bytes to be added to the location counter. Otherwise, the assembler prints an expression mode error (e) and the line is ignored. The comment field is optional. Note that the symbol is assigned the value of the location FOLLOWING the block.

```
BEND    BES 20
```

BSS - BLOCK STARTING WITH SYMBOL

Syntax definition:

{<label>} BSS <exp> {<comment>}

The BSS directive advances the location counter according to the value in the operand field. The label field symbol is assigned the old location counter value before it is updated.

The operand field contains a well defined, absolute expression that represents the number of bytes to be added to the location counter. Otherwise, the assembler prints an expression mode error (e) and the line is ignored. The comment field is optional. Note that the symbol is assigned the value of the location at the BEGINNING of the block.

```
BUFFER  BSS >100        ;GET BUFFER
        DATA 0
```

(11.2.8.3 LOCATION COUNTER DIRECTIVES continued)

EVEN - WORD BOUNDARY

Syntax definition:

{<label>} EVEN {<comment>}

The EVEN directive moves the location counter to the next word boundary (even byte) address. If the location counter is already on a word boundary, the line is ignored.

The label field symbol is assigned the location counter value before any adjustments are made. The operand field is not used and the comment field is optional.

Use of an EVEN directive preceding or following a machine instruction or a DATA directive is redundant since the assembler advances the location counter to a word address for those instructions

```
        TEXT -'MESSAGE'
        EVEN
MES1    TEXT 'HELP'     ;ON EVEN BOUNDARY
        BYTE 0
```

11.2.8.4 OUTPUT CONTROL DIRECTIVES

The following directives control the output of the assembler by forcing page throws, turning on and off the listing, and defining extended operation codes. They are PAGE, TITL, LIST, UNL, and DXOP.

PAGE - EJECT PAGE

Syntax definition:

{<label>} PAGE {<comment>}

The PAGE directive causes the assembler to continue the source program listing on a new page. The PAGE directive is usually not printed in the source listing, but the line counter is incremented. However, when a label is used, the current value of the location counter is assigned to the label and the line is printed to the source listing file. The operand field is not used and use of the comment field is optional.

```
SUB1    ...             ;SUBROUTINE #1
        PAGE
SUB2    ...             ;SUBROUTINE #2
```

The assembler automatically pages the source listing after 56 lines are output to the source listing file.

TITL - PAGE TITLE

Syntax definition:

{<label>} TITL <'string'> {<comment>}

The TITL directive supplies a title to be printed in the heading of each page of the source listing. This directive is not printed in the source listing unless the label field is used or there is an error. However, the line counter is incremented.

```
        TITL 'DEBUG PROGRAM, REV 1.0'
        ...
```

When a label is used, the current value of the location counter is assigned to the label. The operand field contains a character string of up to 50 characters delimited by any character. When more than 50 characters are entered between delimiters, the assembler retains only the first 50 characters as the title and prints a truncation error (t). The comment field is optional.

The title is printed on all pages following the TITL directive until another TITL directive is processed. If the TITL directive is the first line of a source program, the title appears at the head of the first page of the listing.

(11.2.8.4 OUTPUT CONTROL DIRECTIVES continued)

LIST - LIST SOURCE

Syntax definition:

{<label>} LIST {<comment>}

The LIST directive restores printing of the source to the LIST file if it has been disabled by the UNL directive. This directive is not printed in the source listing unless the label field is used, but the line counter is incremented.

When a label is used, the current value of the location counter is assigned to the label. The operand field is not used and use of the comment field is optional.

UNL - NO SOURCE LIST

Syntax definition:

{<label>} UNL {<comment>}

The UNL directive disables the output of the source listing to the LIST file. This directive is not printed in the source listing unless a label is used, but the line counter is incremented.

When a label is used, the current value of the location counter is assigned to the label. The operand field is not used and use of the comment field is optional.

Use of the UNL directive to inhibit printing reduces both assembly time and the size of the source listing.

(11.2.8.4 OUTPUT CONTROL DIRECTIVES continued)

DXOP - DEFINE EXTENDED OPERATION

Syntax definition:

{<label>} DXOP <symbol>,<exp> {<comment>}

The DXOP directive associates a symbol with an extended operation instruction (XOP). When the label field is used, the current value in the location counter is assigned to the label.

```
DXOP OUT,13

OUT @LABEL(R2)
```

The operand field contains a symbol followed by a comma and an expression ranging from 0 to 15. The comment field is optional.

The assembler assigns the symbol to an extended operation specified by the expression. When the symbol appears in the operation field of a line, the assembler inserts the defined XOP as the opcode for that line. However, the assembler maintains only one symbol for each XOP at any one time. If there is no comma in the operand field, the assembler prints a missing operand error (X) and the line is ignored.

NOTE: A symbol assigned to an extended operation MAY also be used as a regular label, and the assembler keeps their meanings distinct.

11.2.9.5 LINKAGE DIRECTIVES

The following directives are used to provide linkage information for the LINKer and include optional files in the source stream. They are DEF, REF, and COPY.

DEF - EXTERNAL DEFINITION

Syntax definition:

{<label>} DEF <symbol>{,<symbol>}... {<comment>}

The DEF directive outputs to the object file one or more symbols and entry addresses for reference by to other programs. When the label field is used, the current value of the location counter is assigned to the label.

```
        DEF LBL1,LBL2,LBL3
LBL1    MOV R1,R2
LBL2    EQU $
LBL3    MOV R4,R5
        ...
```

The operand field contains one or more symbols, separated by commas. These symbols must be well defined on the first pass of assembler. Otherwise, the assembler prints an illegal symbol error (S) and the symbol is not output to the object file. The use of the comment field is optional.

The DEF directive causes the assembler to output to the object file a tag character indicating the mode of the symbol, a four character hexadecimal value, and the six character name of the symbol. This information is used by the LINKer utility in combining program modules which are assembled separately.

REF - EXTERNAL REFERENCE

Syntax definition:

{<label>} REF <symbol>{,<symbol>}... {<comment>}

The REF directive outputs to the object file one or more symbols to be resolved by the LINKer utility. When the label field is used, the current value of the location counter is assigned to the label.

```
        REF LBL1,LBL2,LBL3
        DATA LBL1,LBL2
        MOV @LBL3,R0
```

The operand field contains one or more symbols, separated by commas. The comment field is optional.

No arithmetic can be performed on a REF'd symbol. The value appearing in the source listing when an externally referenced symbol occurs is the address of the symbol in the symbol table, and is not the value output to the object file.

(11.2.9.5 LINKAGE DIRECTIVES continued)

The assembler outputs to the object file a '3' tag character followed by the six characters of the name of the symbol. This information is used by the LINKer utility in combining program modules which are assembled separately.

COPY - INCLUDE FROM FILE

Syntax definition:

{<label>} COPY <file> {<comment>}

The COPY directive temporarily switches from the source file to a new file for text inputs. The operand field contains a PDOS file name from which subsequent source statements are to be read. When the label field is used, the current value in the location counter is assigned to the label. The comment field is optional.

```
START   COPY ASM2:SR
        COPY ASM3:SR
        COPY ASM4:SR
        END START
```

The COPY directive leaves the current source file open, opens the COPY <file>, and reads source code from the COPY file until finished. If the file is not found, the assembler prints a copy file error (C) and the source input file remains unchanged.

A COPY file terminates with either an end-of-file, a LINK directive, or an END directive. If an end-of-file is encountered in a COPY file, the assembler closes the COPY file and resumes reading source code from the original source file beginning with the line immediately following the COPY directive.

If a LINK directive is encountered in a COPY file, the assembler closes the COPY file and processes the LINK command as usual. The new file becomes the COPY file.

If an END directive is encountered in a COPY file, the assembler processes the end directive as usual. Both the COPY file as well as the original file where the COPY directive was processed are closed and the assembly process is terminated.

If the COPY directive is encountered in a COPY file, the assembler prints a file error message and the line is ignored; that is, COPY file cannot be nested.

11.2.8.6 CONDITIONAL ASSEMBLY DIRECTIVES

The following directives are used in conditional assembly of source files. This allows various configurations of a program to be assembled from the same sources by changing only a few flags. They are IFZ, IFN, DUP, and OPT.

IFZ - IF ZERO, GOTO SYMBOL

Syntax definition:

{<label>} IFZ <exp>,<symbol> {<comment>}

The IFZ directive causes the assembler to skip source statements if the expression equals zero. When the label field is used, the current value of the location counter is assigned to the label.

The operand field contains a well defined expression and a symbol, separated by a comma. When the expression evaluates to zero, subsequent source statements are treated as comments until a source line is read which contains <symbol> in its label field. Assembly then resumes normally beginning with the line containing <symbol>. When the expression evaluates to a nonzero value, the assembler ignores the directive and assembly continues normally with the next source statement.

If the expression is not absolute, the assembler prints an expression mode error (e), and the line is ignored. If there is no comma, or if the expression is not well defined, the assembler prints an illegal symbol error (S), and the line is ignored.

IFN - IF NONZERO, GOTO SYMBOL

Syntax definition:

{<label>} IFN <exp>,<symbol> {<comment>}

The IFN directive causes the assembler to skip source statements if the expression is nonzero. When the label field is used, the current value of the location counter is assigned to the label.

```
EMUL    EQU 1           ;SET EMULATOR FLAG
        RORG 0
*
LAB1    LI R0,>F000     ;COMMON CODE
        IFN EMUL,LAB2   ;EMULATOR VERSION?
        BL @SUB1        ;N, DO REAL CALL
*
LAB2    ...             ;CONTINUE COMMON
```

(11.2.8.6 CONDITIONAL ASSEMBLY DIRECTIVES continued)

The operand field contains a well defined expression and a symbol, separated by a comma. When the expression evaluates to a nonzero value, subsequent source statements are treated as comments until a source line is read which contains <symbol> in its label field. Assembly then resumes normally beginning with the line containing <symbol>. When the expression evaluates to zero, the assembler ignores the directive and assembly continues normally with the next source statement.

If the expression is not absolute, the assembler prints an expression mode error (e), and the line is ignored. If there is no comma, or if the expression is not well defined, the assembler prints an illegal symbol error (S), and the line is ignored.

DUP - DUPLICATE LINE

Syntax definition:

{<label>} DUP <exp> {<comment>}

The DUP directive causes the assembler to duplicate the next line. When the label field is used, the current value of the location counter is assigned to the label.

The operand field.contains a well defined expression which contains the duplication count. The range of the count is from 0 to 32767. If the expression equals zero, the next line read from the source file is treated as a comment. If the expression is nonzero, the assembler processes the next line read from the source file, <exp> number of times.

If the line to be duplicated modifies the location counter, then the label field on the line must not be used, or the assembler issues a multiply-defined symbol error (M) each time the line is assembled. Use of the comment field is optional.

The DUP directive is useful when the size of a buffer in a program needs to be varied each time the program is assembled. For this example, an assembly time constant is placed in the operand field and a DATA 0 directive immediately follows the DUP directive.

If the expression is not absolute, the assembler prints an expression mode error (e), and the line is ignored. If the expression is not well-defined, the assembler prints an illegal symbol error (S), and the line is ignored.

```
BSIZ    EQU 7           ;SET BUFFER SIZE
*
BUFFER  DUP BSIZ        ;ZERO BUFFER
        DATA 0
        ...
```

(11.2.8.6 CONDITIONAL ASSEMBLY DIRECTIVES continued)

OPT - SET OPTION FLAG

Syntax definition:

{<label>} OPT <char>=<exp>{,<char>=<exp>}... {<comment>}

The OPT directive sets or resets various assembly flags. When the label field is used, the current value of the location counter is assigned to the label.

The operand field contains one or more equations, separated by commas. Each equation consists of an option character, an equal sign, and a well defined expression. The assembler evaluates the expression and places its value in the associated option flag. The comment field is optional.

The assembler resets all option flags to zero at the beginning of each pass. Therefore, in order to select any of the available options, at least one OPT directive with a nonzero expression must be executed by the assembler.

If the expression is not absolute, the assembler prints an expression mode error (e), and the line is ignored. If the option character is not found, or if the expression is not well defined, the assembler prints an undefined symbol error (U), and the line is ignored.

The option flag and character pairs in the assembler are:

Character	Flag	Function
?	QFLG	Assemble if ? nonzero
#	PFLG	Assemble right or left half
C	CFLG	Output checksum tags
L	LFLG	Expanded list options
R	RFLG	Register cross reference
S	SFLG	Punch symbol table to object
X	XFLG	Output XREF to LIST file

NOTE: Currently, QFLG, PFLG, LFLG, RFLG, SFLG, and XFLG are supported by the assembler. Currently LFLG acts like the LIST and UNL directives. These flags are defined as follows:

QFLG (?) Conditional assembly
PFLG (#) Alternate assembly

```
OPT ?=EMUL&1    ;ASSEMBLE EMUL
OPT R=1         ;XREF REG
```

(11.2.8.6 CONDITIONAL ASSEMBLY DIRECTIVES continued)

Any source statement can be prefaced with either a question mark (?) or a pound sign (#). If present, this special character must appear in the first position of the source line, with the label field beginning in the second position. If the first character of a source line is neither a question mark nor a pound sign, the assembler processes the line normally.

```
        OPT ?=FLG       ;SET ? FLAG
        A R1,R2
?       AI R2,BIAS      ;IF FLAG, ADD BIAS
        MOV R2,R3       ;SAVE
?       BL @DEBUG       ;DO DEBUG
        ...
```

If the first character of the line is a question mark, then the assembler checks the option flag associated with (?), namely QFLG. If the value of the QFLG is nonzero (set), the assembler skips over the (?) and assembles the remainder of the line normally. If the flag is zero (reset), the source line is treated by the assembler as a comment.

If the first character of the line is a pound sign, then the assembler checks the option flag associated with (#), namely PFLG. If the value of the PFLG is nonzero (set), the assembler skips over the (#) and assembles the line normally. If the flag is zero (reset), then the assembler looks for another pound sign, (#), within the source line. If a second # is found beyond the first character, then the remainder of the line immediately following is assembled as the line of source code. If a second # is not found, then the entire source line is treated by the assembler as a comment. In other words, for source lines containing two pound signs, if QFLG is zero, the right half is assembled, and if QFLG is nonzero, the left half is assembled.

LFLG (L) List control flag

Currently, LFLG is associated with the LIST and UNL directives. If LFLG is zero, the assembler outputs the source code the the list file. If LFLG is nonzero, the assembler inhibits output to the list file.

```
LFLAG   EQU 1           ;SET LIST BITS
*
        OPT L=LFLAG&1   ;LIST #1        (UNL)
        ...
        OPT L=LFLAG&2   ;LIST #2        (LIST)
        ...
        OPT L=LFLAG&3   ;LIST #1 & #2   (UNL)
        ...
```

Future use of LFLG will allow selecting condensed listings of DATA, BYTE, DUP, IFZ, IFN, ?, or macro functions of the assembler.

RFLG (R) Register cross reference

Whenever the RFLG is nonzero, occurrences of the register labels (R0-R15) are included in the cross reference listing. The assembler does not include registers in the cross reference unless the RFLG is set nonzero. This feature can be turned on or off so that the registers used within a selected portion of the program can be cross referenced.

```
        OPT R=1         ;START REG XREF
        MOV R1,R2
        ...
        OPT R=0         ;STOP REG XREF
        MOV R1,R2
        ...
```

(11.2.8.6 CONDITIONAL ASSEMBLY DIRECTIVES continued)

SFLG (S) Output symbol table to object file

After the END directive has been processed on the second pass and the entry has been output to the object file, the assembler checks SFLG. If SFLG is zero at the end of the assembly, the object file is closed normally and the assembler exits to PDOS. If SFLG is nonzero at the end of assembly, every symbol in the symbol table is output to the object file, as if each symbol defined in the program had been placed in a DEF directive. Undefined and multiply defined symbols are not output; neither are externally referenced (REF) symbols. The symbol table in the object file can be used by future symbolic debuggers.

Since PDOS only loads until an entry tag (1 or 2) is read, and since the symbol table is output after the entry tag, this option does not affect normal PDOS loading.

```
OPT S=1         OUTPUT SYMBOLS
```

XFLG (X) Cross reference to LIST file

If the XFLG is nonzero at the end of the first pass, and if there was no XREF file option specified when the assembly was initiated, and if there was a LIST file option specified, then the assembler performs a cross reference during the second pass and outputs it to the LIST file, instead of to a separate XREF file. This feature cannot be turned on or off, but is determined by the value of XFLG at the end of the first pass.

```
OPT X=1         ;OUT XREF TO LIST
MOV R1,R2
...
```

11.3 LINK - MODULE LINKER

The PDOS LINK utility combines separately generated object modules into a single linked output module. The linker accepts modules that have been generated by ASM, BASIC, a compiler, or a previous partial link.

Symbol resolution

The major function of the LINK utility is to provide symbol resolution for external references and definitions (REF and DEF assembler directives). The LINK program builds lists of DEF and REF tag symbols. These are resolved by matching DEF tag symbols with the REF tag symbols and outputting new overlay tags with the correct values.

EPROM/RAM partitions

Another function of the LINK utility is to define program and data segments to prescribed boundaries for eventual EPROM/RAM partitioning. Program segments are defined by the PSEG assembler directive, and data segments by the DSEG assembler directive. If these directives are not used, the entire object module is tagged as a program segment.

Partial link

Not all REF tags need to be resolved. A partial link is possible, which can then be included in a subsequent linking process.

The LINK utility prompts with an '*' for a command. If a <carriage return> is entered, the following summary of the LINK commands is output to help you:

```
COMMAND              DESCRIPTION

0,<FILE>             OPEN OUTPUT FILE
1,<FILE>             LINK FILE
2{,<FILE>}           LIST UNDEFINED REFS
3{,<FILE>}           LIST MULTIPLY DEFINED DEFS
4{,<FILE>}           LIST LINK MAP
5                    OUTPUT PARTIAL LINK
6{,<ADR>}            OUTPUT OVERLAYS AND START TAG
7                    EXIT TO PDOS
8{,<ADR>}            LIST/SET PSEG BASE ADDRESS
9{,<ADR>}            LIST/SET DSEG BASE ADDRESS
10                   RESTART
11,<FL>,<S>,<P>      LOAD BASIC BINARY MODULE
12{,<M>}             0=NO DSEG, 1=NORMAL, 2=DSEG=>PSEG
13,<FILE>            LIBRARY
14{,DEFAULT}         SET DEFAULT TASK NUMBER
15{,FLAG}            CODE FLAG
```

The following discussion defines the LINK commands in detail:

(11.3 LINK - MODULE LINKER continued)

*0,<file>	OPEN OUTPUT FILE. An output file is specified for linked object.	OPEN OUTPUT FILE
*1,<file>	LINK FILE. The <file> is read and processed to the output file. If a previous input file has been opened, it is first closed.	LINK FILE
*2{,<file>}	LIST UNDEFINED REF'S. All unresolved REF tags are listed to your console. These can be optionally listed to <file>.	LIST UNDEFINED REF'S
*3{,<file>}	LIST MULTIPLY DEFINED DEF'S. All multiply defined DEF's are listed to your console. These can be optionally listed to <file>.	LIST MULTIPLY DEFINED DEF'S
*4{,<file>}	OUTPUT LINK MAP. The full link map, including IDT's, start tag addresses, and module DEF's and REF's, is listed to your console. This can be optionally listed to <file>.	OUTPUT LINK MAP
*5	OUTPUT PARTIAL LINK. The linker system DEF's are dumped to the output file such that they can be recreated in a subsequent link process.	OUTPUT PARTIAL LINK
*6,<addr>	OUTPUT START TAG AND CLOSE. A start tag specified by <addr> is output and the output file closed. If no parameter is specified, a start tag of >0000 is used.	OUTPUT START TAG AND CLOSE
*7	EXIT TO PDOS. All files are closed and the linker exits to PDOS.	EXIT TO PDOS
*8{,<addr>}	SET PROGRAM BASE ADDRESS. A program base address is set for program linking. This applies only to the PSEG assembler directive.	SET PROGRAM BASE ADDRESS
*9,<addr>	SET DATA BASE ADDRESS. A data base address is set for program linking. This applies only to the DSEG assembler directive.	SET DATA BASE ADDRESS
*10	RESTART. All files are closed and all tables set to null.	RESTART

(11.3 LINK - MODULE LINKER continued)

*11,<FILE>,<S>,<P> — LOAD BASIC BINARY RUN MODULE

LOAD BASIC BINARY RUN MODULE. A BASIC binary file is linked to the output file. Since the file does not contain any TI 9900 tags, the linker must generate TI object code. Special DEF tags, symbols, and values are also generated for linking the BASIC module to the runtime executive module R$MODA. The symbols generated are R$DBxx, R$DExx, R$PMxx, and R$PTxx, where xx is defined by LINK 14 command. Each symbol is incremented by one after each BASIC binary module is loaded. The RAM size is specified in 1K byte increments by <S> and gives the values for R$DBxx and R$DExx. The assigned console port number is specified by <P> and is the value of R$PTxx.

*12,<M> DSEG TAG MODE. Once the system parameters are defined, it is desirable that the output module contain only EPROM or program segment data in standard TI object tags. This means that DSEG tags and data which are no longer of any value are dropped for the resulting object code. If the <M> value equals 1, then DSEG information is passed through normally. A zero value disables DSEG information from being output to the output file. A value of 2 translates all DSEG relocation codes into PSEG relocation codes, so that the the application can be loaded into RAM over the PDOS system for debugging purposes.

DSEG TAG MODE

0 = No DSEG
1 = Normal, pass through
2 = DSEG -> PSEG for debugging

*12
DSEG MODE = >0000

*13,<file> LOAD LIBRARY FILE. The specified library <file> is read and processed to the output file. If a previous input file has been opened, it is first closed.

LOAD LIBRARY FILE

*14{<T>} SET DEFAULT TASK NUMBER. The value <T> ranges from 0 to 15, and is used as the task number for the next LINK 11 command (link BASIC module). If no parameter <T> is specified, the current default task number is printed.

SET DEFAULT TASK NUMBER

*14
TASK NUMBER = >0000

(11.3 LINK - MODULE LINKER continued)

*15{,flag} CODE FLAG. The value {flag} is used to enable and disable object output. A 1 enables, while 0 disables all object code to the output file.

```
CODE FLAG
*15
OUTPUT MODE = 0001
```

Example:

```
.LINK/4
LINKER R2.4
*<CR>
COMMAND         DESCRIPTION

0,<FILE>        OPEN OUTPUT FILE
1,<FILE>        LINK FILE
2{,<FILE>}      LIST UNDEFINED REFS
3{,<FILE>}      LIST MULTIPLY DEFINED DEFS
4{,<FILE>}      LIST LINK MAP
5               OUTPUT PARTIAL LINK
6{,<ADR>}       OUTPUT OVERLAYS AND START TAG
7               EXIT TO PDOS
8{,<ADR>}       LIST/SET PSEG BASE ADDRESS
9{,<ADR>}       LIST/SET DSEG BASE ADDRESS
10              RESTART
11,<FL>,<S>,<P> LOAD BASIC BINARY MODULE
12,<M>          0=NO DSEG, 1=NORMAL, 2=DSEG=>PSEG
13,<FILE>       LIBRARY
14{,DEFAULT}    SET DEFAULT TASK NUMBER
15{,FLAG}       OBJECT MODE, 0=OFF, 1=ON
*12,0
WAS >0001
*9,>4000
WAS >0000
*0,MODULE
*1,R$MODA/4
*1,R$MODB/4
*1,R$MODC/4
*1,R$MODF/4
*11,PRGM0,2,1
MODULE SIZE: 260
*11,PRGM1,2,0
MODULE SIZE: 164
*1,TASK02
*1,SUBR#0
*1,SUBR#1
*4,LINKMAP
*6
START TAG = >0000
*7
.
```

11.4 XBUG DEBUGGER

The XBUG debugger is an assembly language debugger which is loaded into memory using the ALOAD utility. XBUG is entered via a 'GO <entry address>'. The first thing XBUG does is alter XOP 12 for breakpoints. Four breakpoints are available as well as memory dumps, single stepping, and various inspect and change modes.

Breakpoint XOP = 12

XBUG prompts with a question mark (?) and accepts single letter commands. These letters are followed by up to three parameters, depending on the command. All numbers are hexadecimal. Parameters are separated by either commas or blanks. The following illustrates how to load and execute XBUG:

```
.ALOAD XBUG
 *IDT=XBUG2.4
 *ABS ADR=>0070
LAST ENTRY ADR=>6000
.GO >6000
XBUG R2.4
?<CR>
A {PC}
B {#}{adr}
C from,to,into
D from{,to}
E base
F from,to,data{+}
G {PC}{,WS}{,SR}
L FILE
M {adr}{,adr}
P {PC}
Q {base},{bits}
R {#}
S
U {unit}
W {WP}
X
Y {SR}
Z
?
```

(11.4 XBUG DEBUGGER continued)

Assemble

A {addr}

The line by line assembler allows TI 9900 assembly language instructions to be assembled and loaded into memory. The address is automatically set to the task's entry point, unless the {addr} parameter is specified.

$TEXT	Load ASCII text into memory
/<adr>	Assemble at new address
+constant	Load decimal constant
%binary numbers	Load binary
>hex address	
$ is PC location	

Breakpoint

B
B #
B #,<ADR>

The breakpoint command lists, sets, and clears the four breakpoints supported by XBUG. Breakpoints may be placed on any non-execute (X) instruction. The number of words for the instruction is automatically calculated when it is set.

The 'B' command with no arguments lists all current breakpoints. The 'B' command followed by a number from 1 through 4 clears that particular breakpoint.

The 'B' command followed by a number and an address sets a breakpoint. The instruction is disassembled and stored in the breakpoint table. When a 'G' (go execute), 'S' (go single step), or a 'X' (Exit with breaks) command is executed, the breakpoint addresses are loaded with an XOP 12 instruction.

When one of the points is executed, the XOP 12 routine in XBUG performs a break, and the XBUG menu is entered. The break comes after the instruction is executed.

Copy memory

C <adr1>,<adr2>,<adr3>
C FROM,TO,INTO

The block of memory from <adr1> through <adr2> is copied into another block of memory beginning at <adr3>.

(11.4 XBUG DEBUGGER continued)

Disassemble

Memory from <adr1> to <adr2> is disassembled to the screen in TI 9900 assembly language format. The instruction address is followed by a colon, the hexadecimal instruction code(s), and the ASCII assembly language instruction. If parameter <adr2> is omitted, only one instruction is disassembled. Striking a character during a block disassembly causes the output to pause for easier viewing.

```
D <adr1>{,<adr2>}
D FROM{,TO}
```

Set disassembly base

The E command sets a base address which is subtracted from all disassembly addresses. This allows easier correlation of the disassembly with the source listing.

```
E <base>
```

Find data

Memory contents from <adr1> through <adr2> are searched for the byte or word of <data>. Byte data is specified by following <data> with a plus sign (+). Otherwise, only words are compared. The addresses within the block that match, <if any>, are listed, with TABs, to the screen. Striking a character during the address listing causes the output to pause for easier viewing.

```
F <adr1>,<adr2>,<data>{+}
F FROM,TO,DATA{+}
```

Go Execute

XBUG begins target program execution with the 'G' command. Optionally, the program counter, workspace, and/or status register can also be specified by using the <adr1>, <adr2>, and <adr3> parameters, respectively. If no parameters follow, the current PC, WS, and SR are used. The breakpoints, if any, are loaded with the XOP 12 trap before control is passed to the target program.

```
G {<adr1>}{,<adr2>}{,<adr3>}
G {PC}{,WS}{,SR}
```

Load file

The 'L' command loads the PDOS file named <file> as if it were simply being run from PDOS. The assembly language program can be typed either 'OB' or 'SY'. The program entry address is printed, but the program is not executed.

```
L <file>
L FILE1
```

(11.4 XBUG DEBUGGER continued)

Memory IAC

Memory can be dumped, inspected, or changed with the 'M' command. One or no arguments enters an inspect and change mode. A <CR> opens or closes a location. A <SP> closes a location (if open) and moves to the next location. a <^I> goes indirect and <^C> returns to the XBUG menu. If two addresses are given, the contents of the memory from <adr1> through <adr2> is displayed in both hexadecimal and in ASCII. Striking a space bar pauses the display for easier viewing.

M {<adr1>}{,<adr2>}
M {BEGIN}{,END}

CRU IAC

The CRU lines are examined and changed by the 'Q' command. The CRU base address is specified by <base>. The number of CRU bits is specified by <bits>. (Default is 16.) The output consists of the CRU address followed by the number of bits being examined and the contents. A <SP> moves forward and <ESC> moves backward. A <^C> exits to XBUG menu.

Q {<base>},{<bits>}

Program Counter

The user program counter is examined and changed with the 'P' command.

P {<adr>}
P {PC}

Register IAC

User workspace registers are dumped with a single 'R' command. A specific register is opened if the 'R' is followed by a register number in hex (e.g. RF displays the contents of register 15). A <SP> moves to the next register and <ESC> to the previous. A <^C> returns to the XBUG menu and a <^I> goes indirect and enters the memory modify mode. (See the 'M' command).

R {#}

(11.4 XBUG DEBUGGER continued)

Single step

S

The 'S' command enters the target program in a single stepping mode. Breakpoints are loaded, along with the PC, WP, and ST registers. The first instruction is disassembled and displayed, and XBUG waits for a command, which consists of a single character. Legal commands are:

<space>	Execute instruction, show next.
^C	Cancel and return to XBUG menu.
^R	Dump registers.
^S	Dump memory snapshot.

The address is followed by a colon and the memory contents of the current PC. The disassembled mnemonics are also displayed along with the contents of the source and destination operands before execution. If a <space> is entered, a temporary break is loaded beyond the instruction, the instruction is executed, and control is returned to XBUG through XOP 12. The results of the operation are then displayed, along with the resulting status register. XBUG then waits for another command in single step mode. All PDOS calls, and all other XOPs, are not stepped through, but execute in real time, transparent to the debugger. The execute instruction 'X' cannot be stepped through; a user breakpoint must be used instead. If a user break is single stepped through, control returns to the XBUG menu.

A control C <^C> cancels single stepping without executing the current instruction and control returns to the XBUG menu. A control R <^R> dumps to the screen the current contents of all the registers and then waits for another single step command. A control S <^S> dumps a block of memory to the screen as a memory snapshot. The limits of the block that is dumped are set by the last previous block limits used with the 'M' command. To set the snapshot limits, simply exit from single step with a ^C, dump the desired range with 'M' (only a few lines of dump is best), and return to single stepping with an 'S'.

(11.4 XBUG DEBUGGER continued)

Set UNIT

The PDOS output unit is set with the 'U' command. All subsequent output is directed to <unit>. This is helpful when spooling debug output to either a printer (via a $TTA driver) or a file using the 'SU' command of PDOS. The 'U' command in XBUG is identical to performing a 'UN' command from PDOS.

```
U <unit>
U 3             ;Output to file
```

Workspace

The user workspace pointer is examined and changed with the 'W' command

```
W (WP)
```

Exit to PDOS w/Breaks

XBUG returns to PDOS with the 'X' command and sets all breakpoints. If the target program is run from PDOS with the 'GO' command, the first encountered breakpoint will return to the XBUG menu.

```
X
```

Status Register

The user status register is examined and changed with the 'Y' command.

```
Y (SR)
```

Exit to PDOS

XBUG returns to PDOS with the 'Z' command without setting the breakpoints.

```
Z
```

(11.4 XBUG DEBUGGER continued)

XBUG example:

```
 .ALOAD XBUG                                          Load XBUG
  *IDT=XBUG2.4
  *ABS ADR=>0070
LAST ENTRY ADR=>6000
 .GO >6000                                            Entry point is at memory address >6000
XBUG R2.4
 ?                                                    List XBUG menu
A {PC}
B {#}{adr}
C from,to,into
D from{,to}
E base
F from,to,data{+}
G {PC}{,WS}{,SR}
L FILE
M {adr}{,adr}
P {PC}
Q {base},{bits}
R {#}
S
U {unit}
W {WP}
X
Y {SR}
Z
 ?A,A000                                              Instant assemble at address >A000
A000: 02E0  LWPI >B000
A002: B000
A004: 2F5B  +>2F5B
A006: A100  +>A100
A008: 04C0  CLR R0
A00A: 04C1  CLR R1
A00C: A040  A R0,R1
A00E: 0580  INC R0
A010: 17FD  JNC >A00C
A012:       /A100
A100: 0A00  +>0A00
A102: 4D45  $MESSAGE
A104: 5353
A106: 4147
A108: 4500
A114:       <^C>                                     Control C breaks to XBUG
```

(11.4 XBUG DEBUGGER continued)

```
?DA000,A010                                                    Disable from >A000 to >A010
A000: 02E0 B000      LWPI >B000
A004: 2F58           XOP *R11,R13
A006: A100           A R0,R4
A008: 04C0           CLR R0
A00A: 04C1           CLR R1
A00C: A040           A R0,R1
A00E: 0580           INC R0
A010: 17FD           JNC >A00C
?PA000                                                         Examine PC
PC=7220 A000
?S                                                          D005single step
A000: 02E0 B000      LWPI >B000                             D005
A004: 2F58           XOP *R11,R13      S=0288  D=2E47
MESSAGE                                          R=2E47    D005
A008: 04C0           CLR R0                      D=3730  R=0000  D005
A00A: 04C1           CLR R1                      D=0009  R=0000  D005
A00C: A040           A R0,R1           S=0000    D=0000  R=0000  2005
A00E: 0580           INC R0                      D=0000  R=0001  C005
A010: 17FD           JNC >A00C                                   C005
A00C: A040           A R0,R1           S=0001    D=0000  R=0001  C005
A00E: 0580           INC R0                      D=0001  R=0002  C005
A010: 17FD           JNC >A00C                                   C005
A00C: A040           A R0,R1           S=0002    D=0001  R=0003  C005
A00E: 0580           INC R0                      D=0002^C
?B1,A012                                                       Set break point
A012: 5E53           SZCB *R3,*R9+
?G                                                             Continue execution
>>BREAK-1
A012: 5E53           SZCB *R3,*R9+
?R                                                             Dump registers
 R0=0000 R1=8000 R2=4C41 R3=5354 R4=2045 R5=4E54 R6=5259 R7=2041
 R8=4452 R9=3D3E RA=3630 RB=3030 RC=0D09 RD=2E47 RE=4F20 RF=3E36
 WP=B000 PC=A012 SR=3005
?F0,2000,7F                                                    Find all >007F constants from >0000 to >2000
 17B8 1BD2
?F0,2000,7F+                                                   Find all >7F constants from >0000 to >2000
 00CF 052E 129F 1789 19A2 1B14 1B18 1B8C
 1BD3
?P                                                             Examine PC
PC=A012
?W                                                             Examine workspace
WP=B000
?Y                                                             Examine status register
SR=3005
```

(11.4 XBUG DEBUGGER continued)

```
?L,SENDM                                                    Load file SENDM
ENTRY ADDR=7220
?S                                                  R=0000  3005ingle step
A012: 5E53          SZCB *R3,*R9+       S=4400  D=3D44  R=3944  D005
A014: 3E20 6475     DIV @>6475,R8       S=C03B  D=4452
?P7220                                              Set PC to address >7220
PC=A014 7220
?S                                                  R=4452  D006ontinue with single stepping
7220: 2FD0          XOP *R0,R15                 D=3E36  R=3E36  0005
7222: 120A          JLE >7238                           0005
7238: 0203 2400     LI R3,>2400                 D=5354  R=2400  C005
723C: 0204 2408     LI R4,>2408                 D=2045  R=2408  C005
7240: 0205 223E     LI R5,>223E                 D=4E54  R=223E  C005
7244: C020 0000     MOV @>0000,R0               D=0000  R=2FDC  C005
7248: 1505          JGT >7254                           C005
7254: 0225 0032     AI R5,>0032                 D=223E  R=2270  C005
7258: 8103          C R3,R4             S=2400  D=2408  R=2408  0005
725A: 14ED          JHE >7236                           0005
725C: D073          MOVB *R3+,R1        S=FFFF  D=0000  R=FF00  8005
725E: 11FA          JLT >7254                           8005
7254: 0225 0032     AI R5,>0032                 D=2270  R=22A2  C005
7258: 8103          C R3,R4             S=2401  D=2408  R=2408  0005
725A: 14ED          JHE >7236                           0005
725C: D073          MOVB *R3+,R1        S=FFFF  D=FF00
?E7220                                              Set disassembly base of >7220
?S                                                  R=FF00  0006ontinue
003C: D073          MOVB *R3+,R1        S=FFFF  D=FF00  R=FF00  8005
003E: 11FA          JLT >0034                           8005
0034: 0225 0032     AI R5,>0032                 D=22A2  R=22D4  C005
0038: 8103          C R3,R4             S=2402  D=2408  R=2408  0005
003A: 14ED          JHE >0016                           0005
003C: D073          MOVB *R3+,R1        S=FFFF  D=FF00  R=FF00  8005
003E: 11FA          JLT >0034                           8005
0034: 0225 0032     AI R5,>0032                 D=22D4  R=2306  C005
0038: 8103          C R3,R4             S=2403  D=2408  R=2408  0005
003A: 14ED          JHE >0016                           0005
003C: D073          MOVB *R3+,R1        S=FFFF  D=FF00  R=FF00  8005
003E: 11FA          JLT >0034                           8005
0034: 0225 0032     AI R5,>0032                 D=2306  R=2338  C005
0038: 8103          C R3,R4             S=2404  D=2408
```

(11.4 XBUG DEBUGGER continued)

```
?M0,A0                                                    Dump memory from >0000 to >00A0
0000:  2FDC 2306 2F7C 159E 2F7C 159E 2FDC 15F0  /\#./¦../¦../\.p
0010:  2F7C 02DE 2F7C 02DE 2F7C 02DE 2F7C 02DE  /¦.^/¦.^/¦.^/¦.^
0020:  2F7C 02DE 2F7C 159E 2F7C 159E 2F7C 159E  /¦.^/¦../¦../¦..
0030:  2F7C 159E 2F7C 159E 2F7C 159E 2F7C 159E  /¦../¦../¦../¦..
0040:  2F9C 189C 2F9C 1890 2F9C 1886 2F9C 18A8  /.../.../..6/..(
0050:  2F9C 1A28 2F9C 1B2A 2F9C 1BA8 2F9C 197E  /..(/..*/..(/...
0060:  2F9C 4D8C 2F9C 595E 2F9C 5954 2FDC 320A  /.M./.Y^/.YT/\2.
0070:  6D2A 63AE 2FDC 170C 2FDC 1736 2FDC 1758  m*c./\../\.6/\.X
0080:  A55A FFFF 2B0D 6200 2FDC 1740 02EF 007D  %Z..+]b./\.@.o.}
0090:  AA00 9B3D 0078 2E64 0080 0180 0E00 0A00  *..=.x.d........
00A0:  0A40 0A80 0AC0 0B00 0019 0001 000A 0014  .@...@..........
?Z                                                        Exit to XBUG monitor
.LT
TASK  PAGE  TIME    TB    WS    PC    SR    BM    EM   CRU   PORT

*0/0   0     3    >7020 >719A >04C2 >0005 >7000 >E000 >0080 >0001
.GO >6000
XBUG R2.4
?S                                                        R=2408  0005
0000: 2FD0          XOP *R0,R15                  D=003E  R=003E  0005
0002: 120A          JLE >0018                                    0005
0018: 0203 2400     LI R3,>2400                  D=1D4A  R=2400  C005
001C: 0204 2408     LI R4,>2408                  D=4780  R=2408  C005
0020: 0205 223E     LI R5,>223E                  D=7122
?Z
.
```