EDT/ASM III

128/512K COCO-3 DISK
EDITOR/ASSEMBLER

PROGRAMMING SYSTEM
W/DEBUG

CER-COMP
5566 RICOCHET AVE.
LAS VEGAS, NEVADA 89110
(702) 452-0632

EDT/ASM III
REFERENCE MANUAL INDEX

EDT/ASM-3 Text Editor

EDT/ASM III ASSEMBLER

EDT/ASM III
REFERENCE MANUAL INDEX

EDT/ASM III Debug Commands

A Short Tutorial on using EDT/ASM III

EDT/ASM III
INTRODUCTION

INTRODUCTION

This manual is written to acquaint the user with the features of the "EDT/ASM-3" TEXT EDITING, ASSEMBLY LANGUAGE PROGRAMMING SYSTEM. It should be noted by the user that this is a complex program and cannot be fully understood with a single reading. Because it can do so much, you will NOT master EDT/ASM-3 in a few minutes. You WILL be able to start using it quite quickly though.

STARTUP PROCEDURES

This program is a M6809 machine language program written for a Color Computer III using the R.S.Disk System. To execute the program using R.S Disk simply load the program using the "LOADM" command Example: LOADM"EDTASM3"<enter>. This will cause the file to be loaded and automatically executed, the program will come up with the program introduction message and then the "READY" prompt. You are now ready to enter commands to the Text Editr. If an error occurs while trying to load the program, check the disk directory to make sure you are using the same file name as on the disk. Those commands that specifically relate to disk operation are listed separately at the end of the text editor command section.

BACKUP PROCEDURES

R.S. Disk: Make a backup copy using the "BACKUP" command.

Put the backup disk in a safe place. **Always use the original disk to load and execute the program.** Should the original disk fail, use the "Backup Disk" you created to restore the original disk. The original disk comes recorded on both sides for your added protection against a disk failure. The only way the original disk should be written to is with a "BACKUP" command using the backup disk you created to restore the original.

RAMDISK & 512K

If your COCO-3 has 512K of memory installed, EDT/ASM-3 will automatically install 2 RAMDISKs as drives 2 & 3. These RAMDISKs can be used the same as normal disk drives only they are much faster. You can use then to: save temporary files, assemble source files and with files larger then the memory buffer. The RAMDISK storage format is compatible with our own RAMDISK program available separately for only $19.95. When using our RAMDISK, files stored in them will be available when you enter or leave EDT/ASM-3 as well as any of our disk programs.

Cer-Comp does not guarantee this software in any way and will not be liable for any damage resulting from its use.

WHAT IS A TEXT EDITOR?

Since EDT/ASM-3 creates normal ASCII files, ANY text material can be edited. That includes BASIC programs, M/L programs and anything else composed of printable characters. You can:

COMPOSE TEXT
STORE TEXT ON DISK OR TAPE
LOAD IT BACK FROM DISK OR TAPE
CHANGE TEXT
ADD TO TEXT
MOVE TEXT AROUND
DELETE SOME OR ALL TEXT
COPY TEXT
APPEND PREVIOUS TEXT
SEARCH FOR STRINGS IN THE TEXT

Additionally, the EDITOR provides:

AUTOMATIC LINE NUMBERING
SCROLLING UP AND DOWN
INSERT TEXT IN A LINE
DELETE TEXT IN A LINE
KEY OVER CORRECTION OF TEXT
DISK DIRECTORY DISPLAY WITHOUT LEAVING THE EDITOR
EDIT MORE TEXT THAN YOU HAVE MEMORY TO HOLD
KILL FILES FROM WITHIN THE EDITOR

So you see that if it can be done to text, EDT/ASM-3 can do it.

WHAT IS AN ASSMEBLER?

The ASSEMBLER portion of EDT/ASM-3 is the part that creates the Machine Language program. It processes the Source file created or edited by the Text Editor and creates a LOADM or CLOADM binary file on either Tape or Disk.

The ASSEMBLER has several directives that enable it to control listing formats, conditional assembly, addressing modes, and include source library files from disk.

Here is a partial list of the functions in the ASSEMBLER:

SET PROGRAM ORIGINS
SET OR EQUATE LABEL REFERENCES
RESERVE MEMORY AREAS
CREATE CONSTANT DATA AND TEXT AREAS
GENERATE FORMATED PROGRAM LISTINGS
CONDITIONAL ASSEMBLY CONTROL
LIBRARY FILE CONTROL
BINARY CODE FILE GENERATION
ASSEMBLY ERROR DETECTION

Additional Keyboard Characters

EDT/ASM-3 has several keyboard characters that are not normally available on the CoCo. Some of the additional keys generate the same characters as the arrow & shift keys did previously. The reason for this is, when editing, which uses the arrow and clear keys, you can still generate these key codes if necessary.

New Keyboard Characters

Clear/0 = \ ($5C shift/clear)	Clear/1 = | ($7C *n/a)
Clear/2 = ~ ($7E *n/a)	Clear/3 = [($5B shift/down)
Clear/4 =] ($5D shift/right)	Clear/5 = ^ ($5E up/arrow)
Clear/6 = _ ($5F shift/up)	Clear/7 = ‘ ($60 *n/a)
Clear/8 = { ($7B *n/a)	Clear/9 = } ($7D *n/a)

EDT/ASM III
TEXT EDITING COMMANDS

DISK TEXT EDITOR
INSTRUCTIONS

DEFINITIONS:

\ - A "Reverse Slash" is displayed when the SHIFT & "@" keys are depressed as a delimiter for the 'SEARCH' & 'REPLACE commands. Also see editor command summary.

( ) - items enclosed within these characters are required by that command to perform correctly.

[ ] - items enclosed within these characters are considered as optional, when used they must be in the required order.

< > - items enclosed within these characters are comments.

Enter - is used to denote the "ENTER" Key and is used to signify the completion of a line entry.

\- - "Dash" is used as a delimiter between line numbers.

<- - Left arrow is recognized as a Backspace.

BREAK - is used for Break control at any time to return to 'READY'. If BREAK is depressed during a line entry or edit, any changes or entries will be ignored.

Any key can be used to stop the present output and it will be resumed upon entry of any key but "BREAK".

All commands can be abbreviated by using the first two characters of the command followed by its normal parameters.

LINE ENTRY:

Enter a line number, followed by a space and text ending with the "Enter" key.

The line buffer is preset to 255 characters and the cursor will not advance past the last character position, nor will it backspace beyond the first character position. Ten characters before the end of line a medium tone beep will be heard and a higher tone beep will be heard at the end of the line. Any time during line entry if an invalid control character key is entered a double low tone beep will be heard.

Entry of a line number over four digits will result in only the last four digits being accepted.

Entry of a line number followed by "Enter" will delete the line previously entered using that line number.

Entry of a new line using a previously entered line number will cause that line to be replaced with the new line.

Entry of a line with a line number between two previously entered line numbers will insert the new line between them.

Printer Requests:

Any time the printer is requested for an operation the status of the printer is checked for ready. If the printer is found to be in a "NOT READY CONDITION", a message to that effect will be displayed and the program will wait for any key on the keyboard to be pressed, except the "BREAK" key. IF the "BREAK" key is depressed the printer output will be aborted. This will allow those users not having a printer to abort an accidental printer request and not hang up the system.

LIST COMMAND

SYNTAX: LIST [line number] (-) [line number]

Entry without line numbers will list the entire file. Entry with a single line number will list only that line. Entry of two line numbers will list from the first line number to the second one. This is very similar the the "Basic" list function.

Example: LIST 100-300<ENTER>

RENUMBER COMMAND

Syntax: RENUMBER [1 digit increment] [starting line #]

Causes the memory file to be renumbered, if no increment is specified a value of 10 is used. If a starting line # is not specified the increment value is used. If the line #s exceed 9999 before the end of file is reached, the increment value is automatically decreased. The resequence is repeated until a workable value is reached.

Example: RESEQUENCE 5 100
Re-sequence the line numbers in the file begin with '100' and increment each line number by '5'.

DELETE COMMAND:

Syntax: DELETE <begin line#>-<end line#>

The delete function allows large segments of the text buffer to be removed without having to enter each line number to be deleted. If no line specifications are entered the user will be prompted as to whether the entire contents of the buffer are to be deleted. This is mainly to prevent the accidental deletion of the text buffer contents.

Example: DELETE 100-199 <Enter>
Remove all the lines in the text buffer between and including lines 100 thru 199.

SEARCH STRING COMMAND:

Syntax: SEARCH [line #](-)[line #]\[string]\

Searches for all occurrences of the string between the delimiters (Shift @). All the lines containing the specified string will be displayed. If the optional start & stop lines is omitted the search will begin at the beginning of the file to the end of the file. If only the starting line# is specified it will search to the end of file.

Example: SEARCH 100-199 \TEST\
List all the lines containing the string 'TEXT' between lines 100 thru 199.

REPLACE STRING COMMAND:

Syntax: RPLACE [line #](-)[line #] \[string]\[string]\

This function will replace all occurrences of the first string between delimiters (SHIFT @) with the second string. If the optional line #'s are not specified the entire file will be used, if only the starting line # is specified only from there to the end of file will be used, and if both start & end line #'s are specified only the lines including them will be used.

Example: RPLACE 100-999 \TEST\TESTER\
This would tell the editor to replace all occurrences of 'TEST' between lines 100 and 999 with 'TESTER'.

LINE EDIT COMMAND:

Syntax: LEDIT [line #]

Causes the line number specified to be displayed and the cursor to be positioned under the first character of the line. The EDIT mode is then entered, see edit functions under 'AEDIT'.

Example: LEDIT 110 <Enter>
Edit line number 100 using the edit functions.

AUTO EDIT COMMAND:

Syntax: AEDIT [line #]

Causes the automatic edit mode to be entered, if the starting line # is specified the edit function will continue from that line until the end or a cancel edit operation character is entered. All the edit commands are the same as LEDIT (line edit). If no change is required on a line press the Down-Arrow key and the next line will be brought up for editing. If the line is to be deleted just enter Shift"Clear".

Example: AEDIT 100 <Enter>
Begin automatic line editing starting at line 100.

EDIT FUNCTION KEYS

FUNCTION	DEPRESS
MOVE CURSOR RIGHT	Right arrow key
MOVE CURSOR LEFT (backspace)	Left arrow key
MOVE CURSOR RIGHT 1 WORD	Clear & Right Arrow
MOVE CURSOR LEFT 1 WORD	Clear & Left Arrow
INSERT 1 SPACE	Shift & Up arrow keys
MULTIPLE CHARACTER INSERT on/off	Shift & @ keys
DELETE 1 CHARACTER	Shift & Down arrow keys
INSERT WORD (8 spaces)	Clear & Up Arrow keys
DELETE WORD (right)	Clear & Down Arrow keys
MOVE CURSOR TO END OF LINE	Shift & Right arrow keys
MOVE CURSOR TO BEGIN OF LINE	Shift & Left arrow keys
GOTO NEXT SEQUENTIAL LINE	Down arrow key
GOTO PREVIOUS LINE	Up arrow key
END LINE AT CURSOR POSITION	Shift & Clear keys
REPLACE OLD LINE WITH NEW	Enter key
EXIT FROM EDIT MODE	Break key

COPY LINES COMMAND:

Syntax: COPY (from line#)-(to line#) (new location line#)

The copy function allows portion of the current text buffer to be copied to another portion of the file. The lines included in the specifications 'from' and 'to' are copied to the new location line following the destination line. The portion of the file copied is left intact and the file is automatically renumbered upon completion of the copy.

Example: COPY 1100-1345 100
This would place a copy of the lines from 1100 thru 1345 following line 100 .

MOVE & DELETE LINES COMMAND:

Syntax: MOVE (from line#)-(to line#) (new location line#)

The MOVE command works almost exactly the same as the 'COPY' function only the original lines 'from-to' are removed from the file after they are copied to the new location. The file is renumbered the same as in the copy function.

Example: MOVE 1100-1345 100
This would move the lines from 1100 thru 1345 to the next line following line 100.

AUTOMATIC LINE NUMBER COMMAND:

Syntax: AUTO [1 digit increment value] [line #]

Causes the computer to type sequential line numbers incremented by the specified 1 digit value. If not specified the line # will be incremented by 10. Also an optional starting line # can be specified. This is used for entering sequential text lines without having to specify line numbers, they will automatically be typed after each line is entered.

Example: AUTO 100
Enter auto line typing beginning with line '100' with a default increment value of '10'.

MEMORY SIZE COMMAND:

Syntax: SIZE <Enter>

Displays the amount of memory in use, followed by the amount of memory remaining in the text buffer.

PRINTER OUTPUT COMMAND:

Syntax: PRINTER [command line]

Specifies that the next output operation will be output to the printer. Another command may follow the PRINTER command for ease of use. If you want a printed listing of the memory file, this command must be used prior to the LIST command, ex: PR LIST <enter>

Example: PRINTER NLINE LIST<ENTER>
This would tell the editor to list the file to the printer with no line numbers.

EXIT TO BASIC COMMAND:

Syntax: EXIT <Enter>

Causes control to return to 'BASIC'. Once EDT/ASM-3 is exited you cannot return or re-execute the program, it must be re-loaded from disk. If an output file is open the rest of the file will be written to the disk before returning to Basic.

NEW FILE COMMAND:

Syntax: NEW <Enter>

Causes the memory file buffer to be cleared and all pointers reset to the cold start condition. All previously entered information will be lost. You will be prompted with the message "ARE YOU SURE?", if you enter any character other than a "Y" the command will be ignored. If an output file is open the rest of the file will be written to disk before performing the new function.

PRINTER BAUD RATE COMMAND:

Syntax: BRATE <value> Set Printer baud rate

This command will allow users having printers that run at baud rates other than 600 baud, to change printer rates while under EDT/ASM-3 control. The baud rates are set by entering a value from zero thru seven (0-7) to represent the desired rate. The rate values are as follows: [redacted], 0=300, 1=600, 2=1200, 3=2400, 4=4800, 5=9600. 6=19200

Example: >BR 5<enter> Set baud rate to 4800 baud

PRINTER LINE FEED COMMAND:

Syntax: LF<enter> Allow line feed character output

This function is for those users having printers that do not automatically line feed upon receipt of a carriage return character. Normally line feed character output is inhibited, once this command is entered they will be output for each line and cannot be inhibited once enabled.

AUTOMATIC KEY REPEAT DELAY COMMAND:

Syntax: RDELAY <value>

This command allows the user to program whether or not to allow the keyboard keys to automatically repeat and if so, how fast or often it is repeated. If the command is followed by a value of "0" then automatic repeat will be disabled entirely. If a value between 1 and 47 follows, that value will be used to determine how fast the keys will repeat. The smaller the number the faster the key will repeat. The default value is around 15 which causes a repeat at a reasonable rate. Each individual will have to set this to their own personal taste. The delay from the first time a key is pressed until it begins to repeat is approximately 2 seconds and is not adjustable.

Example: RD 5 <enter> Set Repeat Delay to 5 (fast)
RD 0 <enter> Turn Auto Repeat off

SCREEN WIDTH (Characters per line)

Syntax: SW <value>

The SW command allows the user to set the number of characters displayed per line on the Screen. This can be varied from 32 to 80 characters per line in defined steps. The default display comes up in 80 character mode by 24 lines at program startup time, but can be changed to one of 8 different formats. The following values correspond to the number of display characters per line.

```
1 = 32 (192)  5 = 32 (225)
2 = 40 (192)  6 = 40 (225)
3 = 64 (192)  7 = 64 (225)
4 = 80 (192)  8 = 80 (225)
```

The numbers in the parenthesis represent the number of vertical scan lines used on the display. The 225 mode gives an extra pixel width between lines so that the decenders on characters will not appear to touch the tops of the letters on the line below. It makes for a little better display. If your TV or Monitor can't handle the extra lines, select one of the 192 line modes.

```
Example:  SW 8 <enter>  Set width to 80 chars/line (225)
          SW 3 <enter>  Set width to 64 chars/line (192)
```

SCREEN COLOR SELECT:

Syntax: SCREEN <Foreground> <Background>

This command allows the user to select the Foreground (character color) and Background colors for the display. The program defaults to Black characters on a Buff Background (0,63). You can select any color you like from 0 to 63, see page 297 of your COCO-3 manual for some sample color values.

Example: SC 63 0 <enter> BUff chars/Black Background
SC 18 0 <enter> Green chars/Black Background

CHANGE COLOR/MONOCHROME MODE:

Syntax: CColor <enter>

This command allows the user to force the computer to supress the color output to the display or to Enable the color output. By default the program automatically select Monochrome mode when first started up.

Example: CC <enter> Change screen color

TAPE FILE LOAD COMMAND:

Syntax: TLOAD (file name) <Enter>

This is the tape file load command and is used to load 'ASCII' formatted 'BASIC' files or files previously saved by the editor. The file name can be omitted and if so it will attempt to load the next file on the tape. If an error should occur or an attempt to load an invalid file type an error message will be displayed and the load aborted. Note that quotation marks are not used around the file name.

Example: TLOAD TEXT1 <Enter>

TAPE FILE SAVE COMMAND:

Syntax: TSAVE (file name) <Enter>

The TSAVE command is used to save the contents of the current text buffer in an 'ASCII' formatted tape file. Here again the file name can be omitted but is recommended for ease of file identification. The output file is fully compatible with the 'BASIC' tape format and can be reloaded with the BASIC 'CLOAD' command.

Example: TSAVE TEXT1 <Enter>

TAPE FILE APPEND COMMAND:

Syntax: TAPPEND (file name) <Enter>

The TAPPEND command allows a tape file to be appended to the end of the current text file in memory. The file name can be omitted and if so it will attempt to load the next file in the tape.

Example: TAPPEND TEXT2 <Enter>

TAPE FILE SKIP/CHECK COMMAND:

Syntax: SKIP <file name>

This command will allow the editor to search for and skip over tape files much the same as BASIC does. If an error is encountered while reading a file an error message will be displayed and the tape stopped. This can be useful for checking tape files as well as positioning tape for file additions. If a file name is used with the command it will read tape until the file name is found, when it is found the file will be read and the tape stopped at the end of the file. If no file name is used it will simply skip the next file on the tape.

Example: SKIP DEMO$ <Enter>
This would tell the editor to skip past the file DEMO$.

DISK FILE SAVE COMMAND:

Syntax: SAVE [file name.extension:disk drive]

The SAVE command writes the file with the specified file name to disk. If no disk drive/id is entered a default drive of "0" is assumed. The file extension is assumed to be a "DAT" file if not specified. The entire file is saved from the text buffer. If the output file is already in use from a previous file that was larger than the text buffer an error message of 'OUTPUT FILE ALREADY IN USE' will be displayed.

Example: SAVE BIOIA.ASM

DISK FILE OPEN COMMAND:

Syntax: OPEN [file name.extension:disk drive]

The OPEN command opens a disk file for OUTPUT, it is mainly used when you are working on a file that will fit in the buffer and you are running out of memory while trying to move, copy or add text to the file. This will allow you to open up a file and ROLE part of the text out to give you more space to work with.

Example: OPEN DATAFILE:2 <enter>

DISK FILE LOAD COMMAND:

Syntax: LOAD [file name.extension:disk drive]

The LOAD command opens a disk file for input to the text buffer, if line numbers are not included in the text file they will be added. If the file is larger than the available text buffer the user will be prompted for an output file drive and name. If an output file cannot be opened the input file will be closed and only that portion of the file will be accessable for editing. When a duplicate output file is encountered it is automatically removed by the R.S disk system so be aware when specifying file names.

Example: LOAD BIOIA:3

Open the file BIOIA on drive #3 for input and read it into the available text buffer.

DISK FILE APPEND COMMAND:

Syntax: APPEND [file name.extension:disk drive]

The APPEND command adds the file to the end of the present memory file. The Disk drive and file extension options are the same as the'LOAD' command. If the input file is already in use an appropriate error message will be displayed.

DISK DRIVE DEFAULT

Syntax: DRIVE <number>

The Drive command allows you to specify a default disk drive for Disk commands. The value can be in the range of 0 to 65, this allows Hard Disk users to use up to a 10 Meg. drive.

Example: DRIVE 3

ROLL BUFFER OUT TO DISK

Syntax: ROLL <ending line #>

This function is used with files that are larger than the available text buffer. When editing such a file, this function allows the user to write a portion or all of the current text buffer out to the new disk file. If an ending line number is not entered the entire buffer contents are written out to the file. After writing that portion of the file another portion of the input file is read into the text buffer for editing. If the end of the input file is found the user will be notified by the message ' INPUT FILE CLOSED '.

Example: ROLL 2000

This would tell the editor to write only those lines from 0001 to 2000 to the output file and read the next portion of the input file.

GET MORE TEXT FROM FILE

Syntax: GET

This function allows the user to input the next portion of the input file to fill the rest of the buffer. It would be used when a portion of the current buffer was deleted or to get the next line of the input file a single line at a time if the buffer is already full. This would allow the user to get the next few lines of a file to complete a logical block for an assembly language program for editing. No parameters are used in the command line.

END OF FILE EDIT

Syntax: END

This function notifies the editor that the user is finished with the current file for editing. If the input file has not been completely read it will be appended to the current text buffer contents and the rest of the file will be written to the disk and both the input and output files will be closed after all data from the input file has been written to the output file.

DISK DIRECTORY DISPLAY COMMAND:

Syntax: DIR <drive number>

The DIR command allows the user to examine the directory on a specified disk drive. If the drive number is not specified a default drive of "0" is assumed. The disk directory is displayed the same as if the command had been executed from basic and the "Shift @" must be used to pause the display during this command.

Example: DIR 2

This would list the entire directory from the disk on drive number two.

KILL DISK FILE COMMAND:

Syntax: KILL [file name.extension:disk drive]

The KILL command allows you to remove unwanted files from the specified disk. It works basically the same as the Basic "KILL" command except the file extension will automatically default to a "DAT" extension. If not specified the disk drive will automatically default to drive "0". Any errors will be reported the same as normal disk errors.

Example: KILL BIOIA.TXT:3

Remove the file BIOIA.TXT from the disk on drive number 3.

ASSEMBLE FILE COMMAND:

Syntax: ASMB [file name.extension:disk drive]

This command causes the Assembler mode to be entered. Optionally a disk file can be specified for assembling instead of using the text buffer. If not specified it assembles the text buffer as it normally would except that there cannot be an input disk file open, if so an error message will be displayed and control returned to the editor.

Example: ASMB DEMO.TXT:1
Example: ASMB (assemble text buffer)

The first examples would be used to assemble a file from disk with the name "DEMO.TXT" on drive #1. The last example would be used when you want to assemble the file in the text buffer.

CER-COMP EDT/ASM-3 ASSEMBLER OPERATION

The EDT/ASM-3 assembler will assemble 6809 assembler source code and generate executable binary object code either to tape or disk. It will also cross assemble standard 6800 assembler source code to 6809 code compatible object code. These instructions assume that the user is familiar with assembly language programming and, in particular, the language of the M6809 Microprocessor.

Source Code:

EDT/ASM-3 will assemble source files from either the text editor memory buffer, disk files or both (using the LIBrary directive). If the files do not contain line numbers, they will automatically be generated by the assembler. The source line format for EDT/ASM-3 is the same as the standard Motorola Assembler which contains a LABEL (optional), OP CODE, OPERAND, COMMENT (optional). If the line contains a LABEL, it must begin in the first column of the line, the op code is seperated from the label by at least one space character. If a label is not used on the line, the OP CODE or Directive must begin in the second column position. Each element in a source line, must be seperated from the preceeding element by at least one space character, this is what is known as a free form assembler format. Example:

```
0010  NAM TEST
0020  ORG $1000
0030 START LDA #1000
0040  ASLA
0050  END
```

Notice that the first space following the line number is not significant, this is because line numbers, when inserted or deleted automatically remove the additional space. Therefore the label "START" begins in column one and a single space is used to seperate the label from the op code "LDA", no matter if the label is 2 characters or 6 characters long. Also notice that the Directives NAM, ORG and END begin in the second column position since there is no label preceeding them, this is the same format as an OP CODE line in a source file.

There are several Assembler Programming books available for the 6809 processor, we suggest that if you are not familar with the 6809 processor and its instructions, you should use one of the books available from Radio Shack and other sources

Arithmetic Operators & Number Bases

The following operations are permitted during assembly time, which means that an expression is evaluated during the assembly and thus becomes a part of the program being assembled. Numbers may be expressed in one of three bases, which is specified by a special character for Hex and Binary. The default is decimal.

+	Addition
-	Subtraction
*	Multiplication
/	Division
$	Hexadecimal, numbers 0-9 & A-F may follow
%	Binary numbers 0 or 1 may follow

All operations are evaluated from left to right in the order in which they appear. All operations will be converted to 16 bits and truncated to 8 bits for required instructions.

Register Specification:

The 6809 has 9 registers that are accessible to the programmer, four of which are 8-bits and the other five are 16-bits. They are referenced in this assembler by the following notation:

(8) A	Accumulator A
(8) B	Accumulator B
(8) CC	Condition Code register
(8) DP	Direct Page register
(16) X	'X' index register
(16) Y	'Y' index register
(16) U	User Stack pointer
(16) S	System Stack pointer
(16) PC,PCR	Program Counter & Program Counter RELATIVE

The Program counter (PC) can be used to instruct the assembler to assemble code in a Position Independent manner when used in the Indexed mode. When the Program Counter is referred to as 'PCR' it instructs the assembler to determine the offset from the current PC to some absolute address, thus making the code executable anywhere in memory.

```
EXAMPLE:   LEAX MSG1,PCR
```

This would determine the difference between the current PC and the absolute address of MSG1 and use it as the offset for the PC register to calculate the effective address.

```
EXAMPLE :  LEAX MSG1,PC
```

This would use the absolute address of MSG1 to add to the PC register and use that as the effective address to be loaded in the X-reg. This code is not position independent.

Assembler Directives:

Beside the standard machine language mnemonics EDT/ASM-3 supports several Directives. They are instructions for the assembler only and most of them do not assemble into code. The same format applies to these directives as the normal op codes. Brief explanations are given for the directives supported by EDT/ASM-3.

ORG	define new origin (PC=)
END	signal the end of the source file
RMB	reserve memory bytes
FDB	form double byte
FCC	form constant character
FCB	form constant byte
EQU	assign value to symbol
SET	re-assign or assign value to symbol
SETDP	set a value for the DP register
PAG	skip to top of next listing page
SPC	skip specified number of lines
NAM	specify program name (must be first line of program)
OPT	set or reset assembler options
ATH	define author line contents, printed at bottom of page
LIB	include a disk library file
DRV	set default disk drive number
IF	conditional assembly test (true)
IFN	conditional assembly test (false)
ELSE	conditional assembly option
ENDIF	end of conditional assembly segment

ORG <expression,value>

The ORG directive causes a new origin address to be used for the code which follows the directive (PC = address). The value may be a number or a label that has been previously referenced in the source file. It can not be a reference to a label that is later defined in the program.

```
Example:    0010  ORG $1000
```

END <execution address>

The END directive tells the assembler that the end of the source input file has occured. The END directive also allows for the assignment of a starting execution address for the binary disk file if created. The execution address will default to the first byte of code produced by the assembler if not specified. A label can be used for the execution address if previoulsy defined in the program.

```
Example:   0100 START LDA #55
              :
              :
           0250  END START
```

RMB <expression,value>

This directive causes the assembler to reserve memory for variable or data storage. No code is produced only the address counter is changed, it should not be used in the middle of a program except if the object is generated to disk, a tape file will not handle the address change correctly as Basic does not have a provision to handle it.

```
Example:   0040 XTEMP RMB 2 temp storage for ix
```

FCB <value,value,value,etc.>

This directive causes an expression to be evaluated and the resulting least significant 8 bits is stored in memory or generated within the object file. Multiple values may be used to generate several bytes of data each one seperated by a comma.

```
Example:   1120 COLORS FCB $11,$22,$33,$55
           1120  FCB 55 lines per page constant
```

FDB <value,value,value,etc.>

This directive is essentially the same as the FCB directive only the expression is evaluated to 16 bits of data or 2 bytes for each expression.

```
Example:   1120 DECIML FDB 10000,1000,100,10,1
           1130  FDB $1B44
```

FCC <delimiter text string same delimiter>

This directive is used to create strings of characters in the object file for messages or lookup tables etc. Each character in the text string uses one byte of memory space. The two allowable formats are: a count followed by a text string in which case if the string is less than the count specified it is filled with spaces. The second form is where a text string is used by enclosing it between two characters (delimiters) that are the same character.

```
Example:    1390 ERRMSG FCC "AN ERROR HAS OCCURRED"
            1400  FCC :This is a valid "string":
```

label EQU <expression>

This directive is used to equate a symbol or label to an expression or value, no code is generated. A label must be used and an expression or value must follow the directive.

```
Example:    1230 LINEND EQU $04
            1240 PROGEND EQU ENDADD-BEGIN
```

label SET <expression>

This directive is used to set a symbol or label to an expression or value, much the same as the EQU directive. The difference is that a symbol may be SET to differert values within a source file while a symbol may be EQUated only once. The current value of a symbol is the last value SET. A label is required and an expression or value must follow the directive.

```
Example:    1230 DSKFLG SET 1
                  :
            2020 DSKFLG SET 0 no more disk data
```

SETDP <page value>

This directive is used to tell the assembler which memory page it is to use for direct page addressing. The default value is set to 00 for 6800 compatibility. The SETDP directive does not generate any code to alter the DP register value of the processor, it only affects the range of memory that the assembler allows for direct page addressing. For example if "SETDP $10" is used, the range of memory from $1000 thru $1FFF will be selected. Which means that the assembler will select the direct addressing mode of any instruction that accesses this range of memory as long as the SETDP value is not changed. This directive can be used as often as desired within a program, but remember that you must generate the correct assembler code to alter the processors DP register separately.

```
Example:    0100  SETDP $0F set DP range to $F00-$FFF
            0110  LDA #$0F set DP=$0F
            0120  TFR A,DP
```

NAM <file name & comment>

This directive is used to assign a title to the assembler listing and is also used for the disk or tape file name. It must be the first line of any program and can only be used once in the program file.

```
Example:    0010  NAM DISKSORT.BIN disk sort routine
```

SPC <value>

This directive is used to tell the assembler to space down the specified number of lines in the output listing.

```
Example:    1490  SPC 3 space down 3 lines
```

PAG

This directive tells the assembler to skip to the top of the next page in the output listing. If the "NOPAG" option has been set the directive will be ignored.

Example: 4000 PAG skip to next page, new section

ATH <name,comment>

This directive will allow the author or a comment line to be printed at the bottom of each page in the output listing. Any text string following the directive up to 50 characters can be used for the Author line.

Example: 0025 ATH Ralph Smith "CODER"

OPT <specifiers>

The OPT directive determines how and if an object code file is to be generated. There are two options for generation object code, either one or both may be specified they are:

D - Disk binary file creation.
O - Object tape binary file creation.
G - Generate data for FCC,FCB and FDB (default)
S - List symbol table after listing
P - List assembled data in page format (default)
L - List assembled data (default)

These options can also be reset by the use of the "NO" option, that is to reset the Listing option you would simply use "OPT NOL". For a Tape or Disk file to be closed correctly the option must be set at the END directive.

Example: 0020 OPT NOG

LIB <disk file name.ext:drive>

The LIB or LIBRARY directive is used to allow a disk "library" source file to be included in the assembly of the source file currently being assembled. When encountered during assembly, the assembler reads source lines from the specified file until the end of file or an END statement is encountered. It will then resume assembling on the next line following the LIB directive. The LIB directive may be used within an IF/ELSE conditional statement to selectivly include Library source files. LIBrary files may also be nested up to 9 levels, that is a LIB'ed file may call another LIB file up to 9 levels deep.

```
Example1:    2300  LIB DISKIO.DAT:3
```

DRV <value>

The DRV directive allows the user to change the default disk drive number from within a source file. The value may be greater than 3 so that hard disk users can access to that storage. The DRV directive can be changed at any time so that the input, output and library files may be called or output to any drive. This give the capibility to assemble almost an unlimited size source file.

```
Example:    0200  DRV 3
```

Conditional Assembly Directives

EDT/ASM-3 supports the use of "conditional assembly", the ability to assemble only the specified portions of a source file. This can be useful for assembling several different versions of a program where specified functions or different routines may be needed for tape or disk versions of a program, etc..

IF <expression>.....ELSE.....ENDIF

The IF directive evaluates the results of the expression that follows it for a True or False condition. If the result of the expression is TRUE (not equal to 0) then the statements that follow the IF directive up until an ENDIF directive will be assembled, otherwise they will be ignored. Reguardless of the results, assembly will resume with the next statement following the ENDIF directive.

The ELSE directive can be used between the IF and ENDIF directives to provide a more flexible confguration. If an ELSE directive is used, it will effectivly divide the IF/ENDIF format into two parts. If the expression evaluated TRUE, all statements following the IF up to the ELSE directive will be assembled, and those following will be ignored. If the expression evaluated FALSE, all statements following the IF directive up to the ELSE will be ignored and those that follow it will be assembled. The ELSE directive effectivly reverses the results of the expression TRUE>FALSE or FALSE>TRUE.

```
Example:    2300  IF TEST
                :
                : If TEST <> 0 these statements assembled
                :
            2420  ELSE
                :
                : If TEST = 0 these statements assembled
                :
            2540  ENDIF
```

There is one other form of the IF/ENDIF directive, the IFN which means "If Not". This functions the same as the IF directive except that the results of the test are reversed. If the expression results are NOT TRUE then the statements that follow are assembled, otherwise they are ignored.

```
Example:    2300  IFN TEST
                :
                : If TEST = 0 these statements assembled
                :
            2420  ELSE
                :
```

```
          : If TEST <> 0 these statements assembled
          :
      2540  ENDIF
```

EDT/ASM-3 Error messages

Error codes are used to flag source statements that are in violation of the rules and restrictions of this assembler. Error messages are output with three asterisks and the word "ERROR" followed by the error message. The line listed under the error message is the line in error. The Assembler error codes are listed below:

NAM used twice in the same program

EQU directive requires a label

Source statement syntax error

Invalid label. (syntax error)

Symbol has been previously defined

Invalid op code or assembler directive

Short relative branch out of range

Address mode not allowed with op code

Byte overflow. Single byte expression converts to >255

Undefined symbol (not in table)

Invalid register for indexed operation

Re-defined symbol (Pass 2 value differs from Pass 1, usually caused by label address being referenced before assignment)

Directive operand invalid

Symbol table overflow (OUT OF MEMORY)

ASSEMBLING WITH EDT/ASM-3

When the source file has been created or loaded by the editor and is ready to be assembled the assembler must be entered by the use of the 'ASMB' command in the editor. The 'ASMB' command may be followed by an optional Disk file name, extension and drive number in the normal "NAME.EXT:DRIVE" format. If a file name is specified, the assembler will assume that it is the file to be assembled and not the file in memory. If the disk file cannot be located or an error occurs, an error message will be displayed. If the file is found, it is opened for input to the assembler and the normal prompt message is displayed. If no file name is specified following the 'ASMB' command, the assembler will assume the memory file is to be assembled and proceed to the normal pass prompt. Once the pass option prompt is displayed the assembler will wait for an input, at this time a printer can be specified to direct error messages (1 pass) or the listing to the printer. The format is 'P' for printer, followed by the normal Pass selection and option. Note that object code will be generated only if the "2" or "3" pass options have been selected. The "1P" pass will clear any previous symbol table and read the source file creating a new symbol table for those file(s). The "1S" pass will keep the present table and add any new symbols to it generated by those files. The "2" pass options require that either the "1" or "3" pass has already been run if any forward references are made in the program, otherwise errors will occur. The "2" pass can be useful to generate a program listing only (2L) or object output only (2D), the "2P" pass will generate both if the output and listing have not been inhibited thru the "OPT" directive. The "3" pass options are the same as the "2" pass options only the 1st pass is automatically run and then the 2nd pass is run with the specified option. This is useful for programs that have already been debugged and are error free.

Pass Options:

1P	- Build new symbol table
1S	- Add to symbol table
2P & 3P	- Generate object and listing for specified options
2T & 3T	- Generate object only for specified options (OPT)
2L & 3L	- Generate listing only
2O & 3O	- Generate tape object file
2D & 3D	- Generate Disk object file

```
Example:    ENTER PASS: 1(P,S); 2/3(T,P,L,D,O)
            >P3L
```

The printer output was selected by the first character 'P' and the "3L" pass was selected for 1st & 2nd pass listing only.

Differences in EDT/ASM-3

EDT/ASM-3 has some differences from the format standard of the 6809 auto decrement. The normal specification is ,-R and ,--R but in this assembler it can also be ,R- and ,R-- which is the same as the auto increment format of ,R+ and ,R++.

Because of the way EDT/ASM-3 translates 6800 to 6809 code the same code can be generated with different instructions, this can be very convienint in many cases, especially to those who have written a large amount of 6800 assembler code. Some examples are listed below.

```
6809          =     6800

LEAX 1,X            INX
LEAX -1,X           DEX
ANDCC #$FE          CLC
ORCC  #1            SEC
TFR A,B             TAB  (CONDITION CODE NOT AFFECTED)
TFR B,A             TBA  (CONDITION CODE NOT AFFECTED)
LDA #22             LDAA #22 (AVOIDS SYNTAX ERRORS )
PSHS B
ADDA ,S+            ABA
```

As you can see it can be helpfull in some instances, also note that in the translation of the 6800 op codes TAB and TBA the condition codes are not affected on the 6809 as they were in the 6800. So if a program is translated which uses conditional branching following either of these instructions a "TST" instruction must be added following it to insure that the proper condition codes are set for the branch instruction.

EDT/ASM-3 supports the forcing of direct or extended addressing by the use of the symbols "<" and ">". Also the listing output will be flagged in the left margin next to the address with a '*' when non-zero direct page addressing is forced. The listing is also flagged with a '>' in the left margin when an Extended Branch or Jump is not necessary.

EDT/ASM III
Debug Module Commands

The DEBUG module is an extension of the EDT/ASM-3 (Editor/Assembler) package. It allows the user to test and debug machine language programs that have been assembled or loaded to memory. To Begin execution of the DEBUG module use the commadn LOADM "DEBUG" from basic and hit the enter key. The DEBUG program module should load from the disk. When the "OK" message appears enter "EXEC" and hit enter. the DEBUG module sign on message should be displayed and a ">" character will be displayed for a command prompt. The DEBUG module will load into memory beginning at location $1000 (hex). It is position independent and may be relocated by the use of the Block Move command. Ex. BM 1000 17FF 5000<enter> This would move it up to memory at $5000 (hex). To execute the relocated version use the GOto command. Ex. GO 5000<enter>.

The DEBUG module commands are similar to the Editor commands in that they can be abbreviated by the first two characters of the command. The command input line is buffered and will recognize the backspace, clear, break and enter keys for easy error free command entry. Each command line is terminated by hitting the "ENTER" key. Output from any command may be temporarily paused by hitting any key and resumed upon hitting another key.

DEBUG MODULE Commands:

ME <address>	Memory examine & change
SB <address> <etc.>	Set and/or display breakpoint
RB <address>	Remove one or all breakpoints
RS <value> <name>	Set and/or display registers
GO <address>	Goto address with stack
DM <begin> <end>	Dump Memory in Hex & ASCII
FM <begin> <end> <byte>	Fill memory with data byte
FI <begin> <end> <byte> <etc.>	Find memory byte sequence
BM <begin> <end> <destination>	Move block of memory
DA <begin> <end>	Disassemble memory file
EX	Exit monitor back to BASIC
IZ	Re-Initalize DEBUG

DEBUG error codes

AD - Address error (begin > end)
CD - Command error
CE - Conversion error on address or data byte

MEmory <address> Memory examine & change

This function allows the user to examine and change the contents of a specific memory locations on a byte by byte basis. When the function is called and a valid hex address has been entered it will display both the address & data contained in that location of memory in hex. If the address was not a valid hex address or none was entered the last address stored in BEGIN1 will be used. Once the address & data are displayed the user can change that byte, and/or move forward or move backward thru memory. If the data is to be changed simply enter the new hex 2-digit value, if for some reason the new value cannot be stored correctly a '?' will be displayed and the next location will be displayed normally. If an up arrow '^' is entered the previous location will be displayed and if a carriage return 'cr' is entered the function is ended. Any other non-hex character will cause the next location to be displayed.

Example:

```
>ME 3FFE<cr>
3FFE 49 .      period advances to next location
3FFF 98 55     hex value changed to 55
4000 27 11?    new value not changed correctly
4001 31 ^      display previous location
4000 27<enter> end function
```

SBreak <address> <etc.> Set and/or display Breakpoints

The BReakpoint function allows the user to set program breakpoints in memory in order to de-bug programs. If no valid address is entered the function simply displays the contents of the breakpoint table. If a valid address was entered and the table is not full a breakpoint (SWI) will be set in memory and the entry set in the table. It then displays all breakpoints set in the table. When a breakpoint is executed in a program and the SWI interrupt jump vector in memory has not been changed a dump of the registers will be displayed on the system console and the original code will be restored in memory removing the breakpoint. Several breakpoint addresses may be entered on one command line.

Example:

```
>SB 1000 13FF 1103    Set SWI at 1000, 13FF & 1103
1476 1000 13FF 1103   Breakpoint table contents
    shows 1476 was previously set & the new ones set.
```

RBreak <address> **Remove one or all Breakpoints**

This function allows breakpoints previously set by the SB' command to be removed individually or all at once. If a valid address was entered and it is found in the break table only that breakpoint will be removed. If no address is entered all breakpoints in the table will be removed. Notice that 'RESET' will not clear the breaktable unless a system inital- -ization is required and that breakpoints encountered in a program are automatically removed if they are contained in the breaktable. Only one breakpoint address can be removed at a time.

Example:

```
>RB 13FF      remove breakpoint at address 13FF
>RB           remove all breakpoints from table
```

RSet <value> <name> **Set and or display register contents**

This function allows the user to change the value contained in a particular processor register on the defined stack. If no value was entered the function simply displays all the system registers and their contents. If a valid hex value and register name were entered the contents of that register will be replaced by the value entered. The registers will then be displayed for visual verification of the change.

Register Names:

```
C- condition code          A- Accumulator A
B- Accumulator B           D- Direct Page register
X- Index register X        Y- Index register Y
U- User Stack pointer      S- System Stack pointer
P- Program Counter
```

Example:

```
>RS 99 A      change the contents of Acc-a to 99
>RS 100 X     change the contents of IX to 0100
>RS           display register contents
```

GO <address> Goto defined address with stack

This function allows the user to resume processing of a program that was interrupted by a breakpoint or other interrupt that caused a system trap entry. If a valid address was entered with the command that address will be placed in the program counter register on the stack prior to calling an 'RTI' return from interrupt.

Example:

```
>GO<cr>      resume program at address contained in stack PC
>GO 1000<cr> begin execution of program at address $1000
```

DMemory <begin> <end> Dump Memory in Hex & ASCII format

The Dump function allows the user to display & examine areas of memory much easier then using the memory examine & change function. the output is formatted with 8 bytes of data per line with the ASCII characters underneath if printable. The contents of memory between the begin and end addresses will be displayed, if either the begin or end address is omitted or invalid the function will be aborted and an error displayed. If the output is directed to the printer the format will be 16 bytes per line followed by the ASCII characters on the same line.

Example:

```
>DM 100 10F          Display memory from $0100 thru $010F
0100 16 00 79 7E 02 4E 44 55
---- .  .  v  ~  .  N  D  U

>?DM 100 10F  Display  $0100 thru $010F on prn.
```

FMemory <begin> <end> <byte> fill memory with data byte

This function allows the user to fill a defined segment of memory with a specific data pattern. All three paramaters must be entered with the command or an error will be reported. This function can be usefull for initalizing memory for a program or filling memory with a SWI (3F) for trying to trap runaway programs.

Example:

```
>FM 400 600 3F     fill memory from 400 thru 600 with 3F
>FM 2000 4000 00   clear memory from 2000 thru 4000
```

FInd <begin> <end> <byte> <etc.> Find byte sequence

This function will allow the user to search a defined segment of memory for a predefined byte sequence. Any number of bytes can be searched for 1,2,3,4,5 etc. depending upon how many are entered. At least the begin end, and 1 byte to search for must be entered or an error will be reported. If the specified string of bytes is found in the range of memory specified the address and data byte of the previous location, search bytes, and the one data byte following the string will be displayed. The search will then continue until the end address is reached, displaying the information for each occurance of the byte sequence.

Example:

```
>FI A000 BFFF A3 90
A746 BD A3 90 5A
AA68 7E A3 90 9F
```

BMove <begin> <end> <destination> Block memory move

This function will move a defined block of memory from one place in memory to another. The begin and end addresses define the block of memory to be moved and the destination address is where it is to be moved to. If any of the paramaters are not entered an error will be displayed.

Example:

```
>BM 1000 1500 6000  Move 1000 thru 1500 to 6000
```

DAsmb <begin> <end> Disassemble memory into assembler format

This function will dis-assemble a specified segment of memory displaying it in an assembler op code format. It will display the address of each instruction, op code, and operand byte(s). All relative branch instructions will also display a '>' followed by the destination address of the branch instruction. This function is not fool proof by any means and some sequences of memory will be decoded as instructions which are really text characters or data bytes. It is only designed to be an aid in debugging and disassembling programs.

Example:

```
>DA A00E A06F   Disassemble from A00E thru A06F

A00E 10CE 03 D7
A012 86   37
A014 B7   FF23
A017 96   71
A019 81   55
A01B 26   52    >A06E    branch destination address
A01D 9E   72
```

EXit Exit the Debug module back to BASIC

This function simply allows the user to exit from the Debug module back into Basic.

IZ Re-initalize monitor

This function simply re-initalizes the monitor to a cold start condition, prior to initalizing all previously set breakpoints will be restored. This can be usefull if some portion of the monitor temporary storage were modified or simply to reset the baud rates to 600 or any other reason the monitor may not be functioning correctly.

EDT/ASM III Disk file description

The original EDT/ASM III disk as provided contains several different files that are not mentioned in the documentation. A list of the file names and a brief description of them follows.

```
EDTASM3 .BIN  -  The Editor & Assembler program
DEMO    .DAT  -  A short demonstration program.
DEBUG   .BIN  -  The run-time machine language debugger
```

DEMO .DAT - This is a short assembly language program to help you get aquainted with both the Editor and Assembler in EDT/ASM III. It is a simple program that inputs a line of text from the keyboard and when the Enter key is pressed, it displays the line of text that was input.

DEBUG .BIN - This is a relocatable, free standing debug (monitor) package that can be very useful for finding problems in machine language programs and working with the computer on the machine language level.

EDT/ASM III

A Short Tutorial on using EDT/ASM III

EDT/ASM III is a Disk based co-resident Text Editor and Assembler. It will allow the user to create, edit and assemble machine language programs with a minimum of effort. Its Text Editor is far superior to the editors found in most Editor/Assembler programs. It contains powerful search and replace functions, extensive editing features, block copy and move functions, and much more. You can easily create and merge files larger than memory into a single file as large as disk. You have the option of assembling programs stored in the text editors buffer, directly from disk, or both with the use of Library files. It is well suited to creating small utility programs as well a extremely large complex programs. The following is a short tutorial to get you aquainted with some of the commands and features of EDT/ASM III.

Load the Editor/Assebler from the original disk by entering the command:

LOADM"EDTASM3"<enter>

Once the program is loaded, it will automatically execute and the Startup & Ready messages will appear. It will also display the amount of free memory available in your system. Whenever the "READY" prompt appears, you are in the Text Editor command mode, any of the commands listed in the text editor section of the manual may be used when this prompt is displayed.

To start with, enter the command:

DIR<enter>

this should display the directory of the EDT/ASM III disk on the screen. You should see a file with the name "DEMO.DAT" listed in the directory. This is a short assembly language demonstration program. To load the file into the editors buffer enter the command:

LOAD DEMO<enter>

you should see the disk drive select light go on and hear the drive start to load the file. When the file is completely loaded into memory, the message "Input file closed" should appear followed by the ready prompt. If you enter the command:

LIST<enter>

you should see the file displayed on the screen. To pause the display, simply press any key on the keyboard, to resume press any other key, if you press the "Break" key the display will be aborted and return you to the ready prompt.

Now remove the EDT/ASM disk from the drive, we won't be needing it any more. Put a fresh formatted disk in the drive and enter a DIR<enter> command, it should show a blank director. Now we are going to save the demonstration program in memory to disk, enter the command:

SAVE DEMO<enter>

the disk select light should come on and you'll hear the drive start to save the file. When the file is finished being written to disk, the message "Output file closed" should appear followed by the ready prompt. Now enter the command:

NEW<enter>

the screen should clear and the startup message should appear the same as when EDT/ASM was first loaded. The editor buffer is now empty and ready to accept a new program.

We will now start to create a new program, this will help to familarize you with using the text editor. We write a program a line at a time by entering a line number, a space and then the assembler source line. An option we have is to let the computer assign the line numbers automatically. The AUTO command instructs the editor to automatically generate line numbers for us.

AUTO [increment size] [line start #]

An option with this command is the size of the increment an where to start numbering. For example AUTO 1 100 will instruct the computer to start the line numbers at 100 and increment each line by one. If we don't use the options it will start numbering at 10 and use 10 as the increment value. For now we will just enter the command:

AUTO<enter>

you should see the line number 0010 appear on the left edge of the screen followed by a space with the cursor flashing. The cursor is now positioned in the first column or position of the source line. Type a space followed by NAM MYPROGRM<enter>. The "NAM" directive is the first line of any program, it is used to tell the assembler what name to assign to the binary program file when it assembles the program to disk or tape. After you pressed the enter key, the next line number in sequence will automatically appear and your ready to enter the next line. Enter in the rest of the lines listed below.

```
0010  NAM MYPROGRM
0020  ORG $1000
```

```
0030 MYLINE LEAX MYMSG,PCR
0040 MYDISP LDA ,X+
0050  JSR ($A002)
0060  CMPA #4
0070  BNE MYDISK
0080  RTS
0090 MYMSG FCC /THIS IS MY FIRST PROGRAM/
0100  FCB 4
0110  END MYLINE
```

If you made a mistake when typing in the lines, don't worry about it now, shortly we will be showing you how to edit a line. After you entered the last line 0110, the next line number 0120 will appear automatically, press the "Break" key to return to the editor. This little program will display the message "THIS IS MY FIRST PROGRAM" on the screen. First lets try to assemble it and see what happens, enter the command:

ASMB<enter>

When the PASS prompt appears enter 3L, this will tell the assembler to do a 1st pass on the file in memory and when finished, do a 2nd pass with a listing on the screen. You should notice that we had an error when assembling. Before the 2nd pass started, it displayed MYDISK=FFFF, this means that the symbol (label) "MYDISK" was used as a reference in the program but was never defined as a label. During the second pass an error message, "Symbol not defined" was displayed just before line #0070, which means that line 0070 was where the error was detected. To go back and fix the error we must first exit the assembler, press the "E" key to exit the assembler.

There are two edit commands:

LEDIT <line#> single line edit
AEDIT <line#> automatic increment edit

The first edit command LEDIT, is used when you want to make changes to a single line. If either command is used without specifying a line #, the first line of the file will be used, in this case line 10. Since we only have one line with an error in it, line 70, enter the command:

LEDIT 70<enter>

The line will be displayed with the cursor positioned in the first column. Since the error is in the last word of the line (MYDISK should be MYDISP) press the Shift & Right arrow keys. The cursor will move to the end of the line, press the Left arrow key (backspace) and the cursor should be under the letter "K" in MYDISK. To make corrections you can delete the character by using

the Shift & Down arrow keys, or in this case you can just type over the letter in error with the correct letter. Just press the "P" key and the error will be corrected. Now to make the change permanent press the "Enter" key, other wise the line will be left unchanged.

If you entered the wrong line number to edit, you can use the Up and Down arrow keys to go to the previous or next line in sequence. After pressing the "Enter" key to make the changes permanent, the cursor will move to the left of the screen on the next line below the edited line. You are now back in the editor command mode.

The second edit command AEDIT, is used when you want to edit more than one line in the file. If we used the command:

AEDIT 70<enter>

to edit the line, the procedure would be the same. The only difference is that when the "Enter" key is pressed to make the change permanent, the next line in the file would automatically be displayed for editing. The only way to get out of the AEDIT mode is to use the "Break" key or if you reach the end of the file.

When using either of the edit commands, all of the edit functions listed under the section EDIT KEY FUNCTIONS are available. For the most part, they are simple to use. In the above editing example we could also used the Right arrow key to move the cursor under the letter "K" or used a combination of the Clear & Right arrow keys (move word right) to move to the beginning of the word "MYDISK", and then use the single character right key (Right arrow). Remember that all of the EDIT FUNCTION KEYS except the ones that use the "Clear" key will automatically repeat when held down, this can be handy when editing.

Now that we have the error corrected, lets add a few more quick lines to make the displayed message a little longer. To add lines to the file, just pick a line number in between the two lines where you want the line to go. If you are adding more than one line between lines you could also use the AUTO command with an increment of one, starting with the next number higher. In this case we want to add two more lines between the line 90 and 100. Enter the command:

AUTO 1 91<enter>

the line number 0091 should appear on the screen. Enter the following two lines.

```
0091  FCB $0D
0092  FCC /HOW'S THAT FOR ASSEMBLER PROGRAMMING/
```

Now if you list from lines 90 to 100 using the command:

LIST 90-100<enter>

you should see that the new lines have been inserted into the file.

Now return to the assembler by using the ASMB command. When the prompt appears enter a "3L" to assemble the file with a listing to the screen. There should not be any errors, if so go back to the editor and correct them before proceeding. Now that we know the program will assemble without errors, we are ready to assemble it to disk. Enter "3D" at the pass prompt. This will tell the assembler to do both a 1st & 2nd pass but to output the binary object file to disk. The file will have the name that we assigned with the "NAM" direcive in line #0010. When the assembler is finished (pass prompt appears), press the "E" key to return to the editor. If you enter a "DIR" command, you should see the file name "MYPROGRM.BIN" listed in the directory. This is the machine language, binary program file that was generated by the assembler. It is a standard LOADM compatible file, so you can load and execute it from Color Basic.

Disk and Library files

Now that we have successfully assembled the program "MYPROGRAM", save the source file to disk using the command:

SAVE MYPROGRM<enter>

After it is saved to disk in its original form, we will modify it so that it can be used as a library file in conjunction with the DEMO program. We are only going to use the message display section of the program for this example. To do this, we will delete the parts of the program that are not necessary. Enter the commands:

```
DELETE 10-20<enter>
110  END<enter>
```

These commands will delete the range of lines listed (10 thru 20), in the second command line we changed the END statement so it would not have a transfer address. Since we are only deleting a few lines, we could also have just entered the line numbers followed by the enter key, for example:

```
10<enter>
20<enter>
```

Now if you list the file you should see that the lines have been deleted and line 110 only has an END statement. The lines that

remain will be our library file. Save it to disk by entering the command:

SAVE MYMESSAG<enter>

When the file is finished being written to disk, do a directory command DIR, the file MYMESSAG.DAT should be listed in the directory. We will now load the original demonstration program using the command:

LOAD DEMO<enter>

After the file is loaded, we will add a line to the program so that it will automatically include the library file we created earlier. To do this add the following to the program:

0340 LIB MYMESSAG<enter>

Since the original library file was saved with the default extension DAT on the default drive, we do not have to specify the extension or the drive number in the LIB directive. Now go to the assembler using the ASMB command. When the pass prompt appears, enter a "3L". Before the first pass is finished, you should see the disk activity indicator light. This is because the assembler must get the library file from disk for both the 1st and 2nd pass. When the second pass starts (the listing displayed), you should see the normal listing go by until the LIB statement is reached. At this time the listing will pause, while the assembler goes out to the disk to get the library file. The listing will resume, but now the listing is for the library file "MYMESSAG". When the library file is finished being read, the assembler will continue with the next line following the LIB directive. If you want to see a printed listing of the assembler output, you can use the "P3L" option at the pass prompt. Make sure the printer baud rate is set correctly if you are using a print rate other than the standard 600 buad. Once you are sure that there are no errors, assemble the complete program to disk by using the "3D" option at the pass prompt. When done go back to the editor by pressing the "E" key.

Now that you have successfully assembled a program using the LIB directive, lets try using the conditional assembly directives IF, ELSE, and ENDIF. Lets set up the program so that it will only include the library file "MYMESSAG" if the value of the symbol MYMSGF is true (not equal to zero). Add the following lines to the program:

```
0015 MYMSGF EQU 0
0335  IF MYMSGF
0341  ELSE
0342  RTS
```

```
0343  ENDIF
```

Now the program is set up so that if the symbol MYMSGF is true, the library file MYMESSAG will be included in the program and the RTS statement between the ELSE and ENDIF directives will be ignored. If the symbol is false, only the RTS statement will be assembled. To see this, goto the assembler using the ASMB command. At the prompt, enter a "3L", notice that the library file was not included in the program, but the RTS instruction was assembled. Go back to the editor and change line 15 so that it will be true (not equal to zero).

```
0015 MYMSGF EQU 1
```

When this is completed, go to the assembler once again (you should know the command by now), and assemble the program. This time you will notice that the library file was included (assembled) and the RTS statement was not. Well this concludes our short tutorial, I hope that it has proved helpful to you. Remember that this is only a very small sample of what you can do using EDT/ASM III, there are many other editing and assembler commands that we have not explored. Use the demonstration programs and explore some of the other commands and features, I'm sure you'll find a lot of other possibilities available.

Thank You
Cer-Comp