M - ZAL

M-ZAL SYSTEM MANUAL

INTRODUCTION SECTION

TEXT EDITOR SECTION

ASSEMBLER SECTION

XI. PHASES OF ASSEMBLY

LINKER SECTION

III. LINKER COMMANDS

APPENDICES

INTRODUCTION SECTION

I. SYSTEM OVERVIEW

Congratulations on your purchase of M-ZAL for your TRS-80. The M-ZAL system is a complete package which will enable you to develop sophisticated Z-80 assembler language programs. The system consists of three programs:

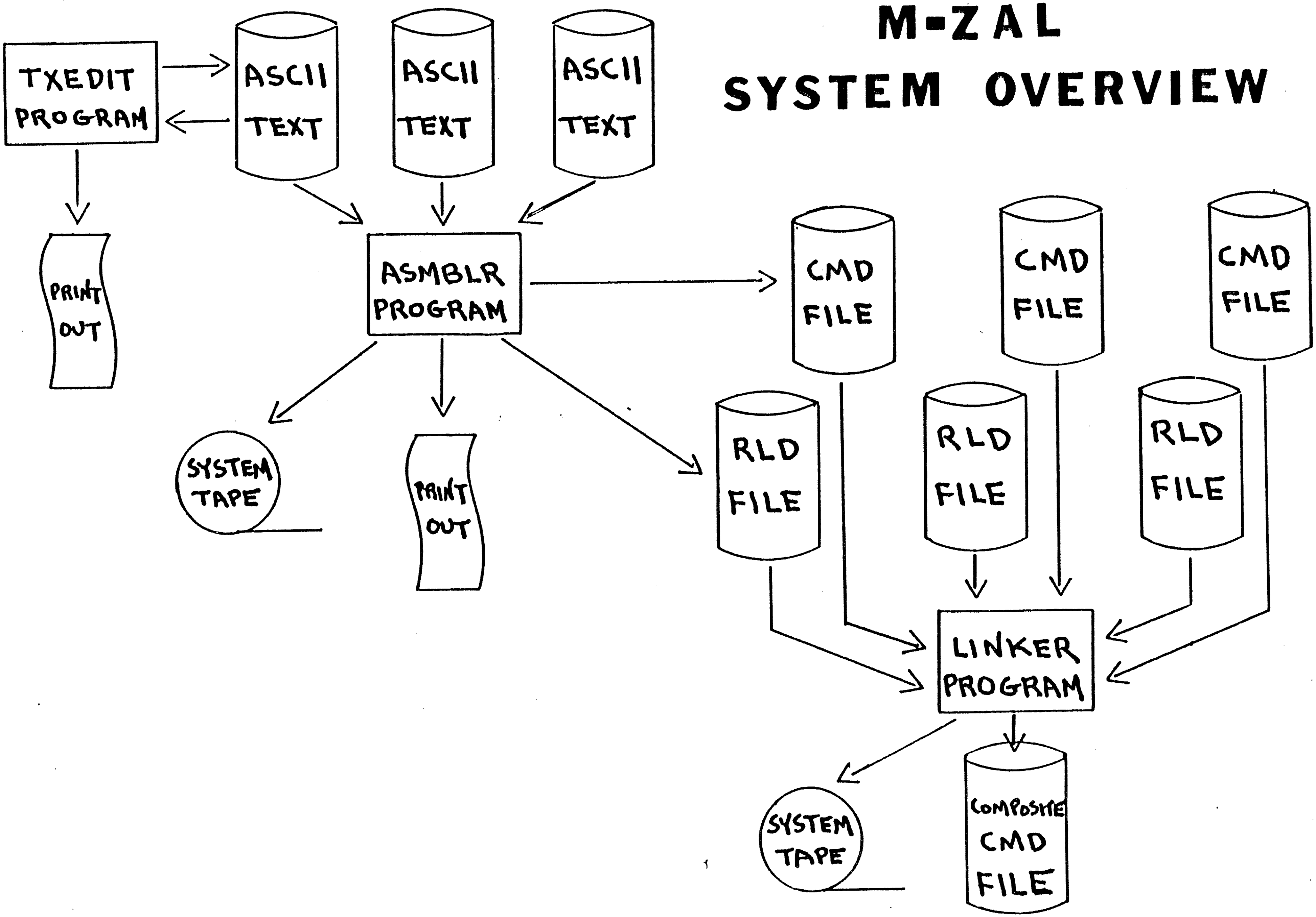
1) TXEDIT - a full screen general purpose text editor which you will use to create, modify, examine, print, and merge disk files containing ASCII text. This program can also be used to convert such disk files to and from cassette tape files in the Radio Shack EDTASM source format.

2) ASMBLR - a menu-driven assembler which accepts as input disk files created by TXEDIT and containing assembler language source programs. ASMBLR can be directed to generate as output any combination of the following: object code to disk as a "/CMD" file, object code to tape as a "SYSTEM" file, listing to the display, listing to the printer, a "/RLD" disk file containing relocation and external symbol information, an alphabetically sorted list of all symbols used in the program, a cross-reference of all symbols used in the program.

3) LINKER - an interactive program that you will use to relocate and link together independently assembled program modules. Through the use of commands entered via the keyboard you will direct the LINKER to read in the "/CMD" and "/RLD" disk files created by the ASMBLR and construct a composite program. The LINKER can then be instructed to write out this composite program, either to disk as a "/CMD" file, or to tape as a "SYSTEM" file.

The accompanying diagram shows the flow of data from one component of the system to the next. This system approach simplifies the development of large and complex assembler language programs by making it possible to break such programs down into smaller and more manageable program modules. Thus the system is named M-ZAL: the Modular Z-80 Assembler Language.

M-ZAL
SYSTEM OVERVIEW



II. THE M-ZAL DISKETTE

The operation of each of the three M-ZAL programs TXEDIT, ASMBLR, and LINKER is discussed in detail in the following sections of this manual. First, however, we will describe the contents of the diskette provided with the M-ZAL system. Unless specially ordered, this diskette does not contain a copy of the disk operating system (DOS). It is therefore necessary to start out by placing a (DOS) system diskette into drive 0 and the M-ZAL diskette into drive 1 of your system. A DIRectory listing will show the following files:

TXEDIT/CMD	the text editor program.
ASMBLR/CMD	the assembler program.
LINKER/CMD	the object module linker program.
TXBASE/RLD	relocation information for the text editor pgm.
DEMO/ASM	a demonstration assembler source text file.
DEMO02/ASM	another demo assembler source text file.
ROMEQU/ASM	ROM routine address equates (to be *INCLuded).
HEXCONV/CMD	hexadecimal conversion subroutine package.
HEXCONV/RLD	relocation & external symbol information for the hexadecimal conversion subroutine package.
DECCONV/CMD	decimal conversion subroutine package.
DECCONV/RLD	relocation & external symbol information for the decimal conversion subroutine package.
EXAMPLE/CTL	a text file containing LINKER control statements.

The first three files listed above comprise the entire M-ZAL system. The rest of the files are provided as examples and demonstrations of the various features of the system and will be referred to throughout this manual.

III. THE MODULE CONCEPT

A central concept of the M-ZAL system is the idea of a program module. A program module is a self-contained assembler language program. Any program module can be represented by up to three different disk files. For example, the program module named ALPHA can be represented by the following files:

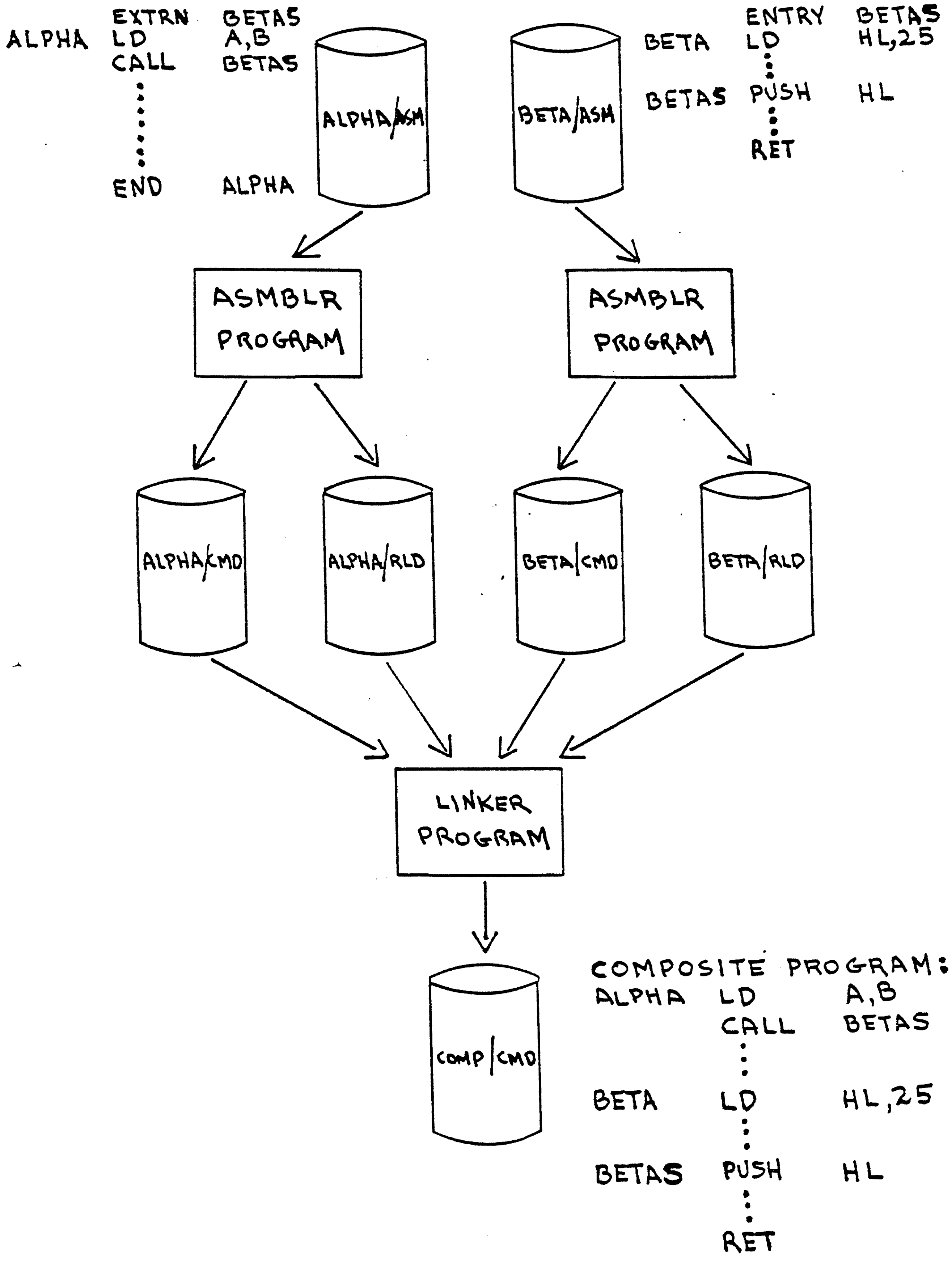
ALPHA/ASM	the ALPHA source program
ALPHA/CMD	the ALPHA object program
ALPHA/RLD	relocation and external symbol information for the ALPHA object program

In practice, the programmer will begin by using the text editor to create the ALPHA source program as a disk file named ALPHA/ASM. The assembler would then be used to "assemble program module ALPHA." This would result in the creation of the ALPHA object program as a disk file named ALPHA/CMD as well as the creation of the third file containing relocation and external symbol information and named ALPHA/RLD. Under DOS, the name ALPHA can be entered as a command and the object file ALPHA/CMD will be loaded and run.

Several different program modules can be created in this fashion and then linked together by using the M-ZAL LINKER program. For example, suppose the ALPHA program just discussed makes reference to a subroutine named BETA5, which is part of another program module called BETA. Each program module is assembled independently and then the LINKER is instructed to load them both into memory. In so doing, the LINKER resolves the reference in module ALPHA to routine BETA5, which is in module BETA. The LINKER is able to accomplish this by using the information provided in each module's /RLD file. The resulting composite program contains the object code of both program modules, with the appropriate cross-referenced addresses filled in by the LINKER. This is illustrated on the following page.

If it is necessary to refer to a program module on a specific disk drive, then the drive designator should be added to the module name (preceded by a colon in a manner similar to the standard DOS approach). For example, the program module named GAMMA:1 refers to the following disk files:

```
GAMMA/ASM:1
GAMMA/CMD:1
GAMMA/RLD:1
```



```
      EXTRN   BETAS
ALPHA LD      A,B
      CALL    BETAS
        :
        :
      END     ALPHA
```

```
       ENTRY   BETAS
BETA   LD      HL,25
         :
BETAS  PUSH    HL
         :
       RET
```

```
COMPOSITE PROGRAM:
ALPHA   LD      A,B
        CALL    BETAS
          :
BETA    LD      HL,25
          :
BETAS   PUSH    HL
          :
        RET
```

TEXT EDITOR SECTION

- The TXEDIT program is a powerful full screen text editor that allows you to create, examine, and modify disk files containing ASCII text. The format of these files is unique to the M-ZAL system and is described in detail in Appendix C. To allow compatibility with Radio Shack's EDTASM product, TXEDIT can read and write EDTASM format source files on tape.

The editor supports variable length lines of text with a maximum length of 128 characters. The largest text file that can be edited is determined by the amount of memory in your system. This does not limit the amount of text that can be assembled at one time, however, because the assembler allows you to chain together several source files in one assembly (this will be explained in the ASSEMBLER SECTION). Each line is assigned a five digit line number by the editor. There is no requirement that the line numbers be kept in ascending order throughout the file, although it it considered good practice to do so.

This section of the manual describes all of the features and facilities of TXEDIT. In order to easily understand the instructions it is strongly recommended that the demonstration source file be loaded from disk and followed during your first reading of this section.

I. INVOKING THE EDITOR

Simply type in TXEDIT in DOS READY mode and the editor will be loaded and run. It will display its initialization frame and ask for a filename. At this point you can type in the DOS filespec for a file containing text that you wish to edit. The file will be loaded and you will be placed in normal edit mode.

Alternatively, if you wish to "start from scratch" and create a new file, you can type in * instead of a filespec. In this case the editor will supply you with a single blank line to start editing with.

At this time you should type in TXEDIT to invoke the editor, and then type in the filespec DEMO/ASM and hit the ENTER key.

II. INTRODUCTION TO EDITING

Now that you have loaded the demonstration source file into memory the format of the editor's display will be explained. The line numbers are in the left 5 columns of the display. Each source line starts a new line on the display and occupies only one line on the display. At present only the first 58 characters of each line can be observed. Obviously only 16 lines at a time can appear on the display. For now imagine that each source line extends past the right of the screen for a maximum of 128 characters (including all blanks) per line and the rest of the lines in the source extend below the screen. You will learn how to "move the display" so you can view the hidden columns and lines. This function will be called scrolling.

The graphics blocks that appear to the right of each line represent unused space on the line. Each line logically ends at the first graphics block. For example line number 40 ends with its last non-blank character but line number 50 ends six characters (blanks) after its last non-blank character. These six blanks are considered significant and are saved when the file is saved. Graphics blocks do not require any space in memory and are not saved with the source.

III. NORMAL EDIT MODE

1) Changing a Character or a String of Characters

When initially entered the editor is in normal edit mode with the first 58 columns of the first 16 lines of text displayed. The first character of the first line will have a non-destructive (blinking) cursor on it.

In order to change a character in this mode move the blinking cursor to the character that you wish to change and type the change. The display will immediately reflect the change and the blinking cursor will advance to the next character. (If the blinking cursor is in the extreme right hand column of the display and a character is entered then the blinking cursor will move to the left hand column of the next line on the display.) Therefore a string of characters can be easily modified and verified for any editing errors. Examples will be given after you are shown how to move the blinking cursor.

2) Moving the Blinking Cursor

The following commands will auto-repeat unless stated otherwise. This will allow you to move the cursor many columns and lines by depressing a command key and holding it down. The cursor cannot be moved to the line numbers on the display. Do not be confused if it appears that the cursor can be moved off of the display. This will be explained in the section on scrolling.

a) → (right arrow)	- move the blinking cursor one column to the right. Ignored if at column 128 of the line.
b) ← (left arrow)	- move the blinking cursor one column to the left. Ignored if at column 1 of the line.
c) ↑ (up arrow)	- move the blinking cursor up one line. Ignored if at the first line in the text.
d) ↓ (down arrow)	- move the blinking cursor down one line. Ignored if at the last line of the text.

e) SHIFT → - move the blinking cursor one tab position to the right. Tab positions are at every eighth character on the line. Ignored if beyond the last tab position (column 120).

f) SHIFT ← - move the blinking cursor to the left margin of the present line on the display. Ignored if the cursor is already at column 1. Does not auto-repeat.

g) ENTER - move the blinking cursor to the left margin of the display on the next line of text. Ignored if the cursor is already on the last line of the text.

Examples

Now that you know how to change a character in the text a few exercises are in order. On the first line of the display the word EDTASM is mispelled as EDTASN. To correct this error depress the → key until the blinking cursor has moved under the N and then type in a M. The mistake is corrected.

On line 70 the comment is preceded by a comma instead of the usual semicolon. To correct this error depress the ↓ key until the blinking cursor has moved under the comma. Type in a semicolon and the error has been eliminated.

Let us assume that we wish to make the comment that we corrected more explicit by changing the word LOC to LOCATION. Use the → key to move the blinking cursor until it is under the position following the C and type in ATION.

By typing over the graphics blocks in the last example the length of the line was increased. If you want to add a word to the comment after LOCATION be sure to type a blank over the first graphics block after the N. If you do not and begin the next word with a graphics block between the two words then the last word will be lost. Remember, the line ends at the first graphics block encountered.

Now that you know how to change a character and move the blinking cursor to any position on the display you might be wondering how to delete a character or how to insert a character on a line.

3) Deleting Characters - SHIFT D (SHIFT↓D)

Move the blinking cursor to the character that you wish to delete. Hit SHIFT D (SHIFT ↓ D on a Model 3) and the character will be deleted. The remainder of the line will be shifted one character to the left to fill the gap that was left when the character was deleted.

This function auto-repeats so that a string of characters can be easily deleted by keeping SHIFT D (SHIFT ↓ D on a Model 3) depressed. This function can only be invoked when you are in normal edit mode (blinking cursor).

Example

On line 50 you will notice that there are six blanks at the end of the line that do not serve any purpose. Because the blanks take up space in memory it would make sense to delete them. Move the cursor to the first blank after the N and depress SHIFT D (SHIFT ↓ D on a Model 3) until the blanks are replaced with graphics blocks.

4) Inserting Characters - SHIFT I (SHIFT↓B)

Move the blinking cursor to the character that you wish to insert characters before and then hit SHIFT I (SHIFT↓B on a Model 3). The editor will go into insert mode and the cursor will stop blinking. A blank will appear where you will be inserting characters. As you type in characters the rest of the line will move to the right.

To exit from insert mode hit the ENTER key. The extra blank will be deleted and the cursor will be blinking again to indicate that the editor is back in normal edit mode.

If you make a typing error while in insert mode use the ← key to backspace the (non-blinking) cursor.

The → key functions differently while in insert mode. When hit, it causes the cursor to advance and blanks to be inserted into the line until the next tab position is reached. This is different from its function in normal edit mode where it causes the blinking cursor to move just one column to the right.

Remember, to enter insert mode you must be in normal edit mode. You know when you are in insert mode because the cursor is not blinking. Once you are in insert mode you cannot do anything that you can do in normal edit mode until you hit the ENTER key to exit from insert mode.

Example

You want to change the comment on line 40 to START OF PROGRAM. Move the blinking cursor to the first character of the comment (P) and hit SHIFT I (SHIFT ↓ B on a Model 3). Type in STARTbOFb (note that "b" represents the SPACE BAR) and then hit ENTER to exit from insert mode. Move the blinking cursor to the blank after the word PROGRAM and then depress SHIFT D (SHIFT ↓ D on a Model 3) until the remainder of the line has been deleted.

5) Scrolling

Now that you are able to manipulate text that appears on the display you must be wondering how to manipulate all 128 columns of a line and how to view all of the lines in the file.

a) Viewing different source lines on the screen

Remember you were told to imagine that the rest of the source lines in the file extend below the display. If you want to modify a line you would move the blinking cursor to that line with the ↓ or ↑ keys. Let us suppose you want to look at line 170 for possible modifications. Move the blinking cursor to line 160. Line 170 is right below line 160 even though you can't see it. How do you get there? Try hitting the ↓ key once. Behold line 170. Because the blinking cursor could not move off of the display the command to move the blinking cursor down one line forced the 16 line window that the display shows to move down one line. Notice that line 10 is not on the display any more but can be imagined to be above the display. Another way to look at it is that if the blinking cursor cannot go to the line then the line will come to the blinking cursor. This function is called scrolling. The commands that move the blinking cursor will also auto-repeat while scrolling.

If you want to view line 10 again depress the ↑ key. Since it auto-repeats the blinking cursor will march up the display until it reaches the top (line 20). It will try to march up further but since it can't it will force the display to scroll one line up and line 10 will appear.

Suppose you want to view lines 170-320. You could depress the ↓ key until these lines scrolled into view. A quicker way to do this is to scroll the display down by 16 lines (called a "page") instead of one line at a time. This can be done by hitting SHIFT↓Z (some of the older Model 1 machines will only require hitting SHIFT↓). Try out this function now. Because this function also auto-repeats you can quickly scan the entire program by depressing SHIFT↓Z. Now scroll back to the beginning of the program (a "page" at a time) by depressing SHIFT↑. Notice that whenever the display is scrolled by an entire page the blinking cursor is placed at the top left margin on the screen.

A command that tries to force the blinking cursor beyond the last line of the text or before the first line of the text will be ignored.

b) Left/Right Scrolling

The concept of scrolling the display up or down applies to scrolling the display left or right as well. At any time only 58 characters of each line are visible on the display. To view the rest of the characters you must scroll right or left. You can see an example of this if you scroll the display to the last page of the demonstration program by depressing SHIFT↓Z until scrolling stops.

Suppose you want to read the rest of the comment that is on line 580 of the program. The rest of the comment can be viewed by depressing the → key until the blinking cursor reaches the extreme right hand column. By keeping → depressed the display will be forced to scroll one column at a time to the right. Note that all the lines on the display are scrolled simultaneously to the same column position. The display can be scrolled back to the left by depressing ← when the cursor is in the extreme left hand column. For fast horizontal scrolling SHIFT → will move the blinking cursor one tab position to the right and once this operation causes scrolling to occur then the function will begin to auto-repeat.

SHIFT ← will force the blinking cursor to the extreme left hand column of the display on the present line and if it is there already then it will cause scrolling to the extreme left to occur so that column 1 is brought back into view.

A command that tries to force the blinking cursor to the left of column 1 or to the right of column 128 will be ignored.

You should now understand how to manipulate any data that is on the display as well as how to move the display to any portion of the file being edited.

IV. LINE COMMANDS

Line commands apply to a specific line or block of lines within the entire text buffer instead of to specific characters within a line. In order to enter a line command the following steps should be taken:

- Move the blinking cursor to the line that the command applies to.
- Hit the CLEAR key. The line will be highlighted to the right of the line number field. This will indicate that you are in line command mode and will show which line you have selected.
- To exit from line command mode at this point (i.e.: if you changed your mind) hit the CLEAR key again. The line will be un-highlighted and the blinking cursor will return to its original position on the line.
- If CLEAR is not hit again as just described then the next key hit will determine the function to be performed.

1) Delete Line - D

Hit D and the highlighted line will be deleted from the text buffer. For example move the blinking cursor to anywhere on line 270. Hit CLEAR. Notice the highlighting to the right of the line number. Hit D. Notice that the line has been deleted and you are back in normal edit mode.

2) Insert Line - I

Hit I and a new line (with a line number of 0) will be inserted after the highlighted line. The new line will contain one blank. The cursor will be in column 1 and will not be blinking. Because the cursor is not blinking you can tell that the editor is in insert mode for the line.

Just type in the desired contents of the new line, hitting → whenever you want to insert blanks to the next tab position. If you want to insert another new line then hit ENTER. The process will be repeated: another new line will be created after the line that you have just typed in and the non-blinking cursor will be in column 1 of it.

After you have typed in the last line that you wish to insert hit CLEAR instead of ENTER. The editor will return to normal edit mode. This is the most important function that you will be using to enter new text so read the following example carefully.

Example

Lets say you want to addd two equate statements to the start of the program (between lines 40 and 50). Move the blinking cursor to line 40 and hit CLEAR. The line will become highlighted and you will be in line command mode for the line. Hit I and a new line will be created between lines 40 and 50. Type in ZERO→EQU→0 and then hit ENTER. Another new line will be created. Type in ONE→EQU→1 and then hit CLEAR. Because you hit CLEAR instead of ENTER no new line will be inserted. Instead you will be returned to normal edit mode.

The lines that you have just added will be used in later examples so make sure that you have followed the above example.

3) Copy Line - C

Hit C and the highlighted line is marked for copying. See the H command for how to complete this function. The line will remain highlighted while the copy is pending.

4) Move Line - M

Hit M and the highlighted line is marked for moving. See the H command for how to complete this function. The line will remain highlighted while the move is pending.

5) Move/Copy Line to Here - H

After you have marked a line for move or copy you will be in normal edit mode. You can move the blinking cursor anywhere in the text (you can scroll the display) and do anything except mark another line (or block of lines, see section on blocks) for a move, copy, delete, or insert.

To complete the move or copy, move the blinking cursor to another line in the text and hit CLEAR to enter line command mode. Then hit H and the line marked for a move or copy will be moved/copied to the point following the line that you have just selected.

While there is a pending move or copy (you have marked the line but have not yet selected where it will go) a graphics block will appear in the upper left hand corner of the display once the display is scrolled. This will remind you that you have a pending move or copy. If you decide that you really don't want to move or copy the line that you have selected then hit CLEAR and then ENTER. The CLEAR - ENTER sequence will cancel any pending move or copy.

Example

Suppose you want to move the two EQU statements that have just been inserted to the end of the program. Move the blinking cursor to the first EQU statement and hit CLEAR and then M. Scroll the display to the end of the program by depressing SHIFT ↓ Z. Notice that the selected line remains highlighted and that the upper left hand corner of the display contains a graphics block to indicate that a move is pending. Move the blinking cursor to line 580 and hit CLEAR and then H. The first EQU statement will appear after line 580. Notice that it retains its original line number (in this case 0). Move the blinking cursor back up to the beginning of the program by depressing SHIFT ↑. Notice that the first EQU statement that you just moved has disappeared. If you had done a copy instead of a move then the EQU statement would now appear in both places (with the same line number). Now move the second EQU statement in the same way.

To copy a line the procedure is exactly the same except that you will select the line to copy with CLEAR and then C instead of M. Remember, to cancel a pending move or copy use the CLEAR - ENTER sequence.

6) Blocks of Lines - B

A block of lines can be deleted, copied, or moved just as easily as a single line. The procedure is exactly the same as for a single line except that a block of lines is selected instead of just one line.

To select the beginning of a block, move the blinking cursor to the line that starts the block and select it for a line command by hitting CLEAR. The line will now be highlighted as it is for all line commands. Hit B and the line is now marked as a block terminator. The next line command (delete, copy, or move) will apply to the entire block as opposed to just the line that you enter the line command on. The next line command can be entered anywhere in the text so scrolling can take place (either up or down) to select the other end of the block. The block is defined as all the lines between and including the line that you marked as the block terminator and the line that you subsequently select for delete, move, or copy.

If a block move or copy was selected you must now move the blinking cursor to another line and then hit CLEAR and then H. This will cause the block to be moved or copied to the point following the line just selected.

As in a one line move or copy, once a block terminator is selected a graphics block will appear in the upper left hand corner of the screen to signify that the block is pending.

A pending block command can be cancelled in the same way that a pending move or copy is cancelled. Just use the CLEAR - ENTER sequence.

Lines or blocks of lines that are moved or copied always retain their original line numbers.

Examples

Because the two EQU statements that you moved to the end of the program earlier are not part of the program they should be deleted. Move the blinking cursor to either of the two EQU statements and hit the CLEAR key and then the B key. That line is now identified as the block terminator. Move the blinking cursor to the other EQU statement and hit CLEAR and then D. The block of two lines has now been deleted.

Suppose you want to copy the three comment lines that are at the beginning of the program to the end of the program (before the END statement). Move the blinking cursor to line 10 or line 30 and hit CLEAR and then B. Move the blinking cursor to the other end of the block and hit CLEAR and then C. Now move the blinking cursor to line 580 using the SHIFT↓Z and ↓ keys. Hit CLEAR and then H. The comments will now appear at the beginning and end of the program.

V. OTHER COMMANDS INVOKED WITH THE CLEAR KEY

There are certain commands that are initiated exactly like line commands except that these commands do not apply to the line that the blinking cursor is currently on. Therefore these commands can be invoked by hitting CLEAR while the blinking cursor is on any line. Each of the following command descriptions assume that you have already hit the CLEAR key.

1) Scroll to Top - T

Hit T and the display will scroll to the top of the text buffer. Getting to the beginning of a very large text buffer is much faster if you hit CLEAR and then T than if you depress SHIFT ↑ . Try getting to the beginning of the demonstration program right now by hitting CLEAR and then T.

2) Repeat Find - F

Hit F and the previously entered FIND command will be repeated. The FIND command will be discussed in the following chapter.

3) Invoke Extended Command Processor - SPACE BAR

Hitting the SPACE BAR will invoke the extended command processor. A command prompt line will appear at the bottom of the display. Extended commands are entered on this line.

VI. EXTENDED COMMAND PROCESSOR

As discussed above, when the CLEAR - SPACE BAR sequence is hit the bottom line of the screen is cleared so that you can enter an extended command. If you change your mind and do not wish to enter an extended command then hit ENTER while this line is empty. You will be returned to normal edit mode. The commands discussed below assume that you have invoked the extended command processor.

Every extended command is terminated by hitting ENTER. You will always be returned to normal edit mode after execution of the extended command entered. The ← key will backspace to allow you to correct any mistakes made while entering the extended command.

1) Renumber - Nline#,increment

To renumber the entire text buffer type in N followed immediately (no spaces allowed) by the value for the first line number, a comma, and then the increment to be used to calculate the rest of the line numbers. Try out this command now. Then use it again as N10,10 so that the line numbers in the demonstration program will be restored to their original values.

It is good practice to always renumber the text before saving it into a disk file. This way, when the text is input to the assembler, each assembly statement will be in order with respect to their line numbers. This makes it easier to use the assembler's symbol table and cross-reference listings, which are keyed to each statement's line number.

2) Search for Line Number - Sline#

To search for a line with a specific line number type in S followed immediately by the line number desired. The search will begin with the line currently at the top of the display and will continue down to the end of the text buffer. If a line with the requested line number is found then the display will be scrolled so as to bring that line into view at the top of the screen. Otherwise a "NOT FOUND" message will be displayed briefly and no scrolling will occur.

3) Find Character String - Fstring

To search the text buffer for a specific string of characters enter F followed immediately by the character string desired. Every character that you type in after the command verb F and before you hit ENTER is considered significant (including blanks). The search will begin with the line currently at the top of the display and will continue down to the end of the text buffer. If the character string is found then the display is scrolled so that the line which contains the found string appears at the top of the screen. (It may be necessary to scroll the display horizontally to bring the string into view.) If the string is not found then the message "NOT FOUND" will appear briefly and the display will not be scrolled.

Remember that the FIND command can be easily repeated by using the CLEAR F sequence.

4) Change Character String - Cstring

When this command is entered the text buffer is searched for the string specified in the previously entered FIND command. Each occurrence of this string is changed to the character string specified in this CHANGE command. This search-and-change operation starts with the line presently at the top of the display and continues down to the end of the text buffer. When the command is completed you will be returned (as always) to normal edit mode - the display is not scrolled.

When entering the CHANGE command remember that every character that you type in is significant (including blanks) up until you hit the ENTER key. Note that because the FIND and CHANGE strings do not have to be of equal length it is possible for lines to grow or shrink in size when they are affected by the CHANGE command. The maximum length of a FIND or CHANGE string is 20 characters.

Examples

You want to find the first source line in the program that uses the DJNZ instruction. Make sure that you are at the beginning of the text buffer by hitting CLEAR T. Then hit CLEAR and then the SPACE BAR to enter extended command mode. Now type in FbDJNZb (note that "b" represents the SPACE BAR) and hit ENTER. The reason you surrounded the DJNZ with blanks is because that is the way it must appear in the assembler program. By searching for it surrounded by blanks you eliminate the possibility of finding an occurrence of DJNZ as part of a larger word. To search for the next occurrence of the same string you can simply hit CLEAR and then F.

You want to change every occurrence of the JR instruction in the program to JP. Get to the top of the text buffer with CLEAR T. Enter CLEAR - SPACE BAR, type in FbJRb and then hit ENTER. The display will be scrolled to the first occurrence of the JR instruction. Now enter CLEAR - SPACE BAR, type in CbJPb and then hit ENTER. All of the JR's in the program are now JP's. Because the lengths of the two strings are the same the column alignment for comments and instructions will not have changed on the changed lines. If you had entered the command CJP instead of CbJPb then the JP would be one column to the left of all the other instructions and the comments on those lines would be 2 columns to the left of all the other comments. This would be due to the fact that you searched for a four character string ("bJRb") and changed it to a two character string ("JP"). Caution should be exercised when using these very powerful commands.

5) Print - P* or Pnumber

The extended command P* will print every line in the text buffer starting with the line presently at the top of the display. To print a specific number of lines enter P followed by the number; for example P13 will print the first 13 lines on the display. Your line printer must be readied before using this command. As each line is sent to the printer its line number is displayed on the bottom of the screen.

6) Load from Tape - Lfilename

Type L followed immediately by an optional filename and hit ENTER. An EDTASM format tape source file will be loaded and appended to the existing text buffer. If a filename is not specified then an un-named file will be loaded. Make sure that your cassette machine is ready and set to "play" before entering this command.

7) Write to Tape - Wfilename

To write an EDTASM format source tape file from the text buffer type in W followed immediately by an optional filename. The text buffer will be written starting with the source line at the top of the display. If you want to write the entire text buffer make sure the display is scrolled to the top first. Make sure that your cassette machine is ready and set to "record" before entering this command.

Note for Model 3 only: the speed (baud rate) at which tape operations take place is determined by a special memory address. Refer to the Radio Shack Model 3 manual for details.

VII. EXITING FROM THE EDITOR (CLEAR - E)

The text editor exit is used when you have finished your editing task and wish to save the results in a disk file. The exit menu also gives you the ability to restart the editor program, return to DOS, or merge the current contents of the text buffer with yet another disk file. The exit menu is invoked from normal edit mode (blinking cursor) by hitting the CLEAR key and then hitting the E key. Do so now.

Notice that the text editor exit menu provides you with five options. Each one will now be discussed in detail.

1) Return to Edit - CLEAR

Hitting the CLEAR key will return you to normal edit mode. The text buffer will be intact and as you left it.

2) Restart the Editor - R

Hitting the R key will cause a restart of the editor. The text buffer will be cleared and the editor's initialization frame will be displayed. The use of this frame was described in chapter I.

3) Return to DOS - D

Hitting the D key will cause the TXEDIT program to terminate and return to DOS READY mode. If you have not previously saved it, the contents of the text buffer will be lost.

4) Save Text Buffer to Disk - S

Hitting the S key will cause the editor to prompt you for the name of the file that you wish to save the current contents of the text buffer into. Type in a valid DOS filespec and hit the ENTER key. The file will be created if it does not already exist and the text buffer will be written out to it. When this operation is completed the exit menu will be refreshed, once again giving you the ability to exercise any of the five options currently being discussed. Note that even after this option is selected with the S key it is possible to abort and return to normal edit mode by hitting the CLEAR key.

5) Add Text from Disk - A

Hitting the A key will cause the editor to prompt you for the name of a file that you wish to add text from. Type in a valid DOS filespec and hit the ENTER key. The contents of the file will be read in and merged with the current contents of the text buffer. This additional text will be inserted into the text buffer immediately following the line that you were on when you invoked the exit menu. This gives you the ability to merge text from another disk file into any point desired in the text buffer. You simply move the blinking cursor to the line after which you wish to insert the text, hit CLEAR E to invoke the exit menu, and then proceed as just described.

When this operation is completed the exit menu will be refreshed, once again giving you the ability to exercise any of the five options currently being discussed. Note that even after this option is selected with the A key it is possible to abort and return to normal edit mode by hitting the CLEAR key.

If the editor encounters an error while reading or writing a disk file then the message "DOS I/O ERROR CODE" will be displayed followed by a two digit hexadecimal number. Refer to your DOS manual for an explanation of the error code. Under certain conditions the editor cannot recover. In these cases hit any key to return to DOS READY mode.

VIII. EDITING SCENARIOS

The example that we have used up to now showed you how to manipulate an existing source text file. Even though you have learnt all the commands needed to build a new source file a review of the procedure should be helpful.

When you enter the editor with * you will be in normal edit mode with the blinking cursor on the first (and only) line. The following procedure can now be used to create a new source file. Many other approaches are possible and after a while you will find one that is most comfortable for you.

Hit SHIFT I (SHIFT↓B on a Model 3) to enter insert mode and type in the first line using → to tab to the necessary columns. After typing in the first line hit ENTER to return to normal edit mode. Now hit CLEAR and then I. You will now be in line insert mode. Therefore you can type in a line of text (using → to get to new tab positions) and then get another new line by hitting ENTER. After you enter the last line hit CLEAR to exit from line insert mode. Now that you are back in normal edit mode you can scroll backwards, checking for and correcting any typing errors. Then renumber and save the source.

You may have noticed that there is no way to copy or move a line to the beginning of the text buffer (before the top line). Instead copy or move the desired line to immediately after the top line and then move the top line around it.

You can repeat lines by doing multiple copies. You can rapidly create a large number of identical lines by doing multiple block copies. This is useful when you are entering part of a program where all of the lines are very similar.

If you follow the rule of always renumbering text before saving it you will always be able to identify which lines were newly added in the current editing session because inserted lines are always given a line number of 0. Similarily, you can always identify the original location of moved or copied lines because they retain their original line numbers when moved/copied.

IX. ERROR MESSAGES

In the event that you request a conflicting or non-existent function the editor will display the message "INVALID REQUEST" for a moment and then you will be returned to normal edit mode.

An example of this would be if you hit CLEAR C to mark a line for copying, scroll around a bit, and then hit CLEAR M. Since you cannot have both a move and a copy pending at the same time you will get the "INVALID REQUEST" message. The move command will be ignored since it was invalid and the copy command will remain pending.

If your program gets very large it is possible for you to run out of available memory. When this happens the editor will display the message "MEMORY FULL" for a moment and then you will be returned to normal edit mode. If at this point you must add more statements to the program then it is recommended that you delete some comments from other lines to make space available. Alternatively, you can split the program into two or more seperate files and link them together at assembly time by using the assembler's *INCL command. This command will be explained in the ASSEMBLER SECTION of this manual.

X. SUMMARY OF COMMANDS AND MODES

NORMAL EDIT MODE

- Change Character - type over blinking cursor.
- Delete Character - SHIFT D (SHIFT ↓ D on Model 3), auto-repeats.
- Insert Character - SHIFT I (SHIFT ↓ B on Model 3), provides non-blinking cursor, → inserts blanks to next tab position. Use ENTER to exit.

MOVING THE BLINKING CURSOR (auto-repeats except where noted)

- ↓ - down one line.
- ↑ - up one line.
- → - right one column.
- ← - left one column.
- SHIFT ↓ Z - down one page (16 lines). Older Model 1 machines do not require the Z key.
- SHIFT ↑ - up one page (16 lines).
- SHIFT → - right one tab position (auto-repeats only when scrolling occurs).
- SHIFT ← - left margin or if already in left margin then scroll to column 1 (does not auto-repeat).
- ENTER - left margin of next line down.

LINE COMMANDS

To invoke hit CLEAR on desired line. Line will be highlighted. To exit without entering any command hit CLEAR again. Otherwise the next key hit determines the function as follows:

- D - DELETE line or block of lines.
- C - COPY line or block of lines.
- M - MOVE line or block of lines.
- B - mark this line as a BLOCK terminator.
- H - move or copy line or block of lines to HERE.

LINE COMMANDS (continued)

- I - enter Line Insert Mode. Cursor not blinking. Puts you in insert mode on a new line. Hit ENTER to get another new line. Hit CLEAR to exit from Line Insert Mode. → will insert blanks to the next tab position in this mode.
- T - scroll display to TOP of text buffer.
- E - invoke TEXT EDITOR EXIT menu.
- F - repeat previously entered FIND command.
- ENTER - cancel pending move, copy, or block command.
- SPACE BAR - invoke EXTENDED COMMAND PROCESSOR.

EXTENDED COMMANDS

Typed in at the bottom of the display. To exit with no command just hit ENTER. All commands are terminated with ENTER.

- Nline#,increment - RENUMBER entire text buffer.
- Sline# - SEARCH for line# from top of display.
- Fstring - FIND character string. Start search at top of current display.
- Cstring - CHANGE previously specified FIND string to this string. Applies to rest of text buffer starting with top of current display.
- P* - PRINT text buffer starting with line at top of current display.
- Pnumber - PRINT text buffer for number of lines specified starting with top of display.
- Lfilename - LOAD the file specified from tape and append to the text buffer. If filename omitted then load an un-named file.
- Wfilename - WRITE the text buffer to tape starting with the line at the top of the current display. If filename omitted then an un-named file will be created.

XI. CUSTOMIZING THE EDITOR

The speed at which the editor's auto-repeating functions repeat can be altered to suit personal taste. It is also possible to modify the editor's use of memory and therefore allow for larger or smaller text buffers. Appendices A and B of this manual describe how these changes can be made. It is necessary, however, to understand how to use the LINKER program in order to make these changes. We suggest therefore that the reader complete the entire M-ZAL manual before exercising this customization option.

ASSEMBLER SECTION

I. INTRODUCTION TO ASMBLR

The M-ZAL System assembler is a very powerful tool that has many uses. This section of the manual describes the operation of the assembler. The reader should be familiar with how to use TXEDIT prior to reading this section. TXEDIT is used to create assembler language source files which then become input to the assembler.

An understanding of assembler language programming and binary number systems is assumed, however, key terms will be explained.

1) File Notation

The assembler uses a structured, modular concept when referencing disk files. When an assembly is performed the assembler is given a "module name" to assemble. The module name is from one to eight alphabetic and numeric characters where the first character must be alphabetic. The assembler constructs disk file names from the module name by adding predefined extensions.

For example, if the assembler is given a module name of SEARCH it will read the source input from the disk file named

SEARCH/ASM

If instructed to output object code to disk it will create a file named

SEARCH/CMD

If instructed to create a relocation/external symbol file, it will create one named

SEARCH/RLD

If it is necessary to reference files on a specific disk drive the disk drive designator should be added directly to the module name. For example, if the assembler is given a module name of SEARCH:1 it will use disk files named

SEARCH/ASM:1
SEARCH/CMD:1
SEARCH/RLD:1

2) Output of the Assembler

The basic use of the assembler is to generate object code for use on the TRS-80. Object code is defined as the binary instructions and data that the Z-80 chip inside the TRS-80 reads and executes. In addition to generating object code the assembler also generates loading instructions which specify where in memory to place the object code and where in memory to start executing it. The term location counter is used to refer to the memory address at which a specific piece of object code will be loaded. The term transfer address is used to refer to the memory address at which program execution will begin.

The assembler is capable of generating object code and loading instructions in two different formats. One of these is the "SYSTEM" tape format. SYSTEM tapes can be loaded and executed by the SYSTEM command in level 2 or disk BASIC. For instructions on how to do so, see your Radio Shack manual. If the assembler is instructed to create a SYSTEM tape, the file name will be taken from the first six characters of the program module name. These tapes can only be created at 500 baud (Model 3 users can use LINKER to create 1500 baud tapes).

The second format is the /CMD disk file. Such a file can be loaded and executed directly from "DOS READY" mode by simply typing in the corresponding program module name. For example, if a program module named TEST3 is assembled and a /CMD file is created, then typing in the command TEST3 from DOS READY will load and execute the program.

It is possible to create both formats of object output with one assembly.

The assembler can also generate a disk file containing relocation and external symbol information for the program in question. This file is given the /RLD filename extension and is therefore referred to as the /RLD file.

The /RLD file and the /CMD file can be used together by the LINKER program to create a new /CMD file or SYSTEM tape. The LINKER can be used to change the loading instructions and object code of a single /CMD file. This function is called relocation. A program that used to load and execute at one memory address can be relocated to load and execute at a different memory address without reassembling the source.

Another function of the LINKER is to merge several program modules together. This is very useful if you have many subroutines that are used by one main routine. You do not have to assemble all of the subroutines together with the main program in order to use make use of them. All you have to do is link them together.

If you are a novice assembler language programmer you may want to use the assembler for a while without generating /RLD files or using the LINKER. After you have become experienced in the use of the assembler you can start using these advanced features. Remember that you do not have to use the LINKER in order to write, assemble, and execute assembler language programs.

II. INVOKING THE ASSEMBLER

The assembler is invoked from "DOS READY" mode by simply typing in the command ASMBLR . The assembler will be loaded and run. It will display the full screen option menu and wait for your commands.

The full screen option menu takes the place of keyword commands. You do not have to memorize any keywords to get full use of all of the assembler's options.

1) Using the Full Screen Option Menu

The full screen option menu always displays each assembler option that can be selected. An X next to an option indicates that it is currently selected. Conversely, if the space next to an option is blank, then that option is currently not selected.

A ? appears on the screen next to the option that can be modified at the present time. The ? can be moved around the screen, from option to option, by hitting the ENTER key.

To select an option, use the ENTER key to move the ? until it is next to the option and then hit the X key. An X will appear next to the option to indicate that it has been selected and the ? will move on to the next option.

To deselect an option use the same procedure except hit the SPACE BAR instead of the X key when the ? is next to the option in question.

When the ? is next to the FILENAME option a blinking cursor will appear. The program module name to be assembled should be typed in next to this option. The → and ← keys can be used to move the blinking cursor back and forth within the module name field. The ENTER key is used to move the ? on to the next option.

For example, if you have created a source file named FRED/ASM:2 using TXEDIT and you want to assemble it, you would type in FRED:2 next to the FILENAME option. Remember that the program module name does <u>not</u> include the "/ASM" extension.

Note that any combination of assembler options is valid. You can therefore select or deselect any option without regard for the states of the other options. When initially loaded, a default set of options is selected. It is therefore only necessary for you to provide a program module name in order to start an assembly.

2) Starting an Assembly

Once the desired options have been selected and a program module name is given, an assembly can begin. An assembly is started by hitting the ↓ key. This can be done at any time, regardless of where the ? is located on the menu.

3) Control during Assembly

Three keys are used to control the assembler while an assembly is in progress. They are:

CLEAR key - the assembly can be cancelled at any time by hitting the CLEAR key. The message "CANCEL" will appear at the bottom of the display. The ENTER key must now be hit to return to the option menu.

SPACE BAR - the assembler can be paused during the time that it is producing the listing on the display and/or printer by hitting the SPACE BAR. Once paused, the assembly can be resumed by hitting the ENTER key.

ENTER key - the ENTER key has several uses:

- If the assembler has paused due to the SPACE BAR, an error message, or a *PAUSE statement then it can be resumed by hitting the ENTER key.
- If the assembly has been terminated by the CLEAR key ("CANCEL") or by a terminal error message then hitting the ENTER key will return control to the full screen option menu.
- If the assembly has terminated normally a message will appear on the listing indicating the number of errors. Hitting the ENTER key will return control to the full screen option menu.
- If OBJECT TO TAPE has been requested then a "READY CASSETTE" message will appear after the listing is produced. Make sure that your tape machine is set up to record and then hit the ENTER key. The assembler will then start writing the object code to the tape.

4) Returning to DOS

You can return to DOS whenever the full screen option menu is active by simply hitting the ↑ key. Once back in DOS, the TXEDIT or LINKER programs can be invoked, as can any other DOS command or utility.

III. ASSEMBLER OPTIONS

All assembler options are displayed on the full screen option menu. An X next to an option indicates that it has been selected. Each option will now be explained in detail.

1) Object to DISK/TAPE

Object code does not have to be generated for an assembly to take place. Object code can be directed to disk and/or tape. If object code is directed to disk then it will be placed in a file whose name is constructed from the program module name and the extension "/CMD". If object code is directed to tape then a "SYSTEM" tape will be created. The "SYSTEM" tape file name will be the first six characters of the program module name.

2) Wait on PAUSE/ERRORS

If the Wait on Pause option is selected then the assembler will pause while producing the listing whenever a *PAUSE statement is encountered. The assembly can be resumed by hitting the ENTER key. If this option is not selected then all *PAUSE statements will be ignored.

If the Wait on Errors option is selected then the assembler will pause while producing the listing whenever a statement which has an error is encountered. The assembly can be resumed by hitting the ENTER key. If this option is not selected then the assembler will not pause when errors are encountered.

3) List to DISPLAY/PRINTER

A listing does not have to be generated for an assembly to take place. A listing can be directed to the video display and/or printer. If neither listing option is selected then the assembler will still make a listing pass but only error messages and the statements that caused them will be listed. All *PAUSE statements will be ignored if no listing option is selected.

4) Symbols LIST/XREF

A symbol table listing and/or xref (cross-reference) does not have to be generated for an assembly to take place. If selected, they will be directed to the device(s) specified by the selected "List to" options. If neither "List to" option is selected then the symbol table listing/xref will be directed to the video display.

The symbol table listing contains each symbol (label) defined in the program. The symbols are listed in alphabetical order. Also provided for each symbol is the line number of the statement that it was defined in, its value, and some of its attributes. If selected, the cross-reference provides the line numbers of each statement in the program that contains a reference to the symbol. Refer to chapter IX for more details.

5) Generate /RLD

A /RLD file does not have to be generated for an assembly to take place. If this option is selected the assembler will create a disk file containing relocation and external symbol information for the assembled program. The disk file name will be constructed from the program module name and the /RLD extension.

The /RLD file and its corresponding /CMD file can be used by the LINKER program to relocate, link, and modify the object program without performing another assembly.

IV. ASSEMBLER LANGUAGE SYNTAX

The M-ZAL assembler recognizes two kinds of source statements: the assembler language statement and the assembler command statement. In this chapter the assembler language statement will be described. Assembler command statements are described in chapter VII.

The format of an assembler language statement is as follows:

```
        label   opcode  operand(s)      comment
```

As you can see, each statement is divided into four fields. Each field is seperated from its neighbors by one or more blanks. The label field must begin in column 1 of the line. The label field and the comment field are optional. The opcode field is always required. The operand field is usually required, though this depends upon the contents of the opcode field.

1) Labels

The label field must begin in column 1 and is optional. If a label is not used then at least one blank must precede the opcode field.

The value of the label is determined by the opcode. If the opcode causes object code to be generated then the label will take on the value of the location counter (memory address) that the object code will be loaded at. This value is always considered relocatable.

Certain pseudo-opcodes do not generate object code. The value of labels associated with such pseudo-opcodes will be discussed in chapter VI.

A label consists of a string of one to eight characters. The first character must be a letter (A - Z). The rest of the characters may be letters (A - Z), numeric characters (0 - 9), or one of four additional characters (@, #, $, and %).

The following characters strings cannot be used for labels as they are reserved to represent Z-80 registers and condition codes:

A, B, C, D, E, H, L, I, R, IX, IY, SP, AF, BC, DE, HL,
Z, M, P, NC, NZ, PE, PO

The following are examples of valid labels:

ALPHA   TEST3   VALUE%   HIGH@VAL   X029

The following are examples of invalid labels:

9TIMES $DATA   IX   R   %INCR   TOOLONGXXX

2) Opcodes

The M-ZAL assembler accepts the standard Z-80 opcodes that are defined in Zilog's "The Z-80 CPU Technical Manual." A copy of this reference has been included with this M-ZAL manual.

The assembler also supports a number of pseudo-opcodes. Pseudo-opcodes do not correspond to Z-80 instructions but instead direct the assembler to generate specific types of object code, change the location counter, give specific values to labels, or terminate the assembly. They are described in detail in chapter VI.

3) Operands

The operands required by an assembly language statement are determined by the opcode. Certain opcodes do not require any operands. Others require one or two operands. If two operands are required then they should be seperated by a comma with no intervening blanks.

Many Z-80 opcodes only allow certain operands to be specified. For example, if an instruction references a one byte register then only the character strings A, B, C, D, E, H, L, and possibly I or R would be permissible as operands. This is fully defined in the enclosed Zilog manual.

Many Z-80 opcodes require operands that specify a one or two byte numeric value. These operands are called "value operands" and are described in detail in the next chapter.

4) Commenting an assembler language source program is considered an essential practice. The comment field must be seperated from the operand field by at least one blank. A semicolon (;) is required before a comment only if

   - the statement's opcode is one which allows zero or more operands (such as the "END" pseudo-opcode), or

   - the entire statement is to be considered a comment. In this case the statement should begin in column 1 with a semicolon.

V. VALUE OPERANDS

As mentioned above, certain operands specify a numeric value. These operands are coded as constants or expressions. (An expression consists of one or more constants on which numeric and/or logical operations are performed.) The range of valid values for a value operand is determined by the instruction it is associated with. For example, in the instruction

```
                LD      HL,nn
```

"nn" can be any two byte value, whereas in the instruction

```
                LD      A,n
```

"n" can be any one byte value. Certain instructions are even more restrictive, for example in

```
                IM      i
```

"i" is only valid as 0, 1, or 2. The format of value operands will now be discussed in detail.

1) Constants

A constant can be expressed as a symbolic label, a numeric value, a quoted character string (also known as a "literal"), or the location counter symbol "$".

a) Symbolic Labels

Every label in the program has set value during the assembly. This value will be either relocatable or absolute, depending on how the label was defined. The syntax of a label has already been defined. Examples:

```
LABEL   EQU     5       ;equate LABEL to constant 5
        LD      A,LABEL ;loads reg A with constant 5
```

b) Location Counter - $

The $ symbol may be used to represent the current value of the location counter. This value will be considered relocatable. Example:

```
            JP      $       ;infinite loop, same as:
  LABEL     JP      LABEL
```

c) Character Strings

A character string constant is represented by one or two ASCII characters surrounded by quotes ('). Its value is always considered absolute. Examples:

```
            LD      A,'B'   ;load reg A with 42H (66D)
            LD      HL,'01' ;load reg HL with 3031H
```

If the quote character itself is to be part of the character string then it should be represented by two consecutive quotes. Note that this is different from Radio Shack's EDTASM syntax. Examples:

```
            LD      A,'''    ;this is an invalid operand
            LD      A,''''   ;load reg A with 27H (')
            LD      HL,'A''' ;load reg HL with 4127H
```

d) Numeric Constants

Numeric constants consist of a string of numbers (0 - 9) or hexadecimal digits (0 - 9, A - F) followed by a constant descriptor. The constant descriptor indicates the base of the constant and can be specified as

B for binary,
O for octal,
D for decimal, or
H for hexadecimal.

If the constant descriptor is omitted then the base of the constant is assumed to be decimal. The first character of any numeric constant must be a number (0 - 9). Therefore hexadecimal values which begin with the digits A - F must be preceded by a 0. Examples:

The following constants are all equivalent:

```
  10        10D     1010B    12O     0AH

            LD      A,10     ;load reg A with 0AH
            LD      HL,000AH ;load reg HL with 000AH
```

Constants that cannot be expressed with 16 bits are invalid and will generate an "INVALID OPERAND" error. An example of such an invalid constant is 0FFFFFH.

2) Expressions

Expressions consist of one or more constants on which numeric and/or logical operations are performed. The assembler evaluates expressions to calculate the operand values that they represent. The resulting value may be considered relocatable or absolute (explained in detail later).

Expressions are always evaluated from left to right, no parenthesis are allowed. Expressions are always evaluated as 16 bit values. If during evaluation a value is generated that cannot be expressed in 16 bits then any overflow is lost (i.e.: 0FFFFH+2 evaluates to 0001H with no error).

The operators that can be used in expressions are:

+	(addition),
-	(subtraction, or negation),
&	(logical AND), and
<	(logical SHIFT).

a) Addition (+)

The + sign can be used as a unary operator. When used in this fashion the constant that it operates on is unchanged. When used as a binary operator, its two operands are added together. Examples:

```
            LD      A,5+0AH  ;load reg A with 0FH
            LD      HL,$+50H ;load reg HL with current loc-
    ;                         ation counter + 50H
```

b) Subtraction (-)

The - sign can be used as a unary operator. When used in this fashion the constant that it operates on is twos complemented. When used as a binary operator, its two operands are subtracted. Examples:

```
            LD      A,5-0AH  ;load reg A with 0FBH
            LD      HL,$-2   ;load reg HL with the current
    ;                         location counter - 2
```

c) Logical AND (&)

The & sign is used only as a binary operator. When used the assembler calculates the logical AND of its two operands. This operator should not be used on relocatable values if you intend to perform relocation. Examples:

```
            LD      A,5&-5        ;load reg A with 1
            LD      HL,0FFFFH&25  ;load reg FL with 0019H
```

d) Logical SHIFT (<)

The < sign is used only as a binary operator. It is used as follows to shift a value left or right:

```
value<amount
```

If "amount" is positive then "value" is shifted left by "amount" number of bits. If "amount" is negative then "value" is shifted right by -"amount" number of bits. Examples:

```
LD      A,5<-3      ;load reg A with 0
LD      HL,5<3+1    ;load reg HL with 0029H
```

3) Relocation Considerations

This section does not have to be read unless you intend to use LINKER to relocate object programs. When an object module is relocated not only are the loading instructions changed, but some of the object code itself is also changed. The following example should make this clear:

```
START   LD      A,0
        LD      HL,START
```

Let us assume that during assembly the label START has the value 7000H. The object code that will be generated for the 2nd statement shown above will specify a relocatable constant value of 7000H to be loaded into reg HL. Now suppose we relocate this program to an starting address 2000H greater than its assembled starting address. The object code labelled START is now located at address 9000H instead of 7000H. The object code for the 2nd statement must therefore be changed to specify a value of 9000H to be loaded into reg HL. This modification of object code is performed by LINKER on every relocatable constant in the program when relocation is performed.

Most object programs will also contain value operands that are not modified in this fashion when relocation is performed. For example, the displacement value "d" in `LD A,(IX+d)` and the constant value "n" in `LD A,n` will not be altered when relocated. It is important to realize that the assembler will allow you to use operands that are normally considered "relocatable" for these values. We advice against such use if you intend to relocate the program because the results can be quite strange. For example, consider the program

```
        ORG     0
START   LD      HL,START ;proper use of relocatable value
        LD      A,START  ;dangerous use of relocatable value
```

As assembled, this program will load reg HL with zero and will then load reg A with zero. If the LINKER is used to relocate it to location 1000H, the result will be a program which loads reg HL with the value 1000H but still loads reg A with a value of zero! The inconsistency has occurred because the relocatable operand "START" was used in an instruction which does not generate a two byte address value. We caution again that M-ZAL does not produce an error message for this situation.

In a similar vein, the assembler allows you to code relative jump (JR) instructions to absolute locations even though JR instructions are always self-relocating. For example, the program

```
                ORG      1000H
        START   LD       HL,0
                JR       1000H    ;jump operand is "absolute"
```

will result in an infinite loop regardless of where it is relocated to.

When an expression is evaluated the assembler determines whether it is relocatable or absolute. If only absolute constants are used in the expression then the expression will be considered absolute. If an equal number of relocatable constants are added and subtracted from the expression then it is still considered absolute. But if the number of relocatable constants that are added into the expression is one more than the number of relocatable constants that are subtracted from the expression then it is considered relocatable. Any other combination results in an "INVALID USE OF RELOCTABLE LABELS" error. Examples:

```
        START   LD      HL,0                    ;absolute
        ALPHA   LD      HL,START                ;relocatable
                LD      HL,ALPHA-START          ;absolute
                LD      HL,ALPHA-START+START    ;relocatable
                LD      HL,START+START          ;invalid
```

4) Linking Considerations

This section does not have to be read unless you intend to use LINKER to relocate object programs. Labels that are defined as external (see pseudo-opcode EXTRN) may not appear in expressions. External labels may be used as operands only in statements which will generate two byte values. Any other use of external labels will cause an "INVALID OPERAND" error. Example:

```
                EXTRN   MULT16
                LD      HL,0
                OR      C
                CALL    NZ,MULT16       ;valid external reference
                RET
                DEFW    MULT16          ;valid external reference
                DEFW    MULT16+0010H    ;invalid
                DEFB    MULT16          ;invalid
```

VI. PSEUDO-OPCODES

Pseudo-opcodes are used in assembly language statements but do not correspond to Z-80 instructions. Labels are always allowed to appear on those pseudo-opcodes that generate object code. The use of labels on pseudo-opcodes that do not generate object code will be discussed for each specific pseudo-opcode of this nature.

1) ORG

This statement takes the form

```
          ORG     value operand
```

and instructs the assembler to set the location counter to the value operand specified. This value will appear in the location counter field of the listing. Notice that no label is allowed on this statement. The value operand is always considered relocatable and may not be an external symbol.

2) EQU

This statement takes the form

```
     label   EQU     value operand
```

and instructs the assembler to set the symbolic label specified to the value of the operand specified. This value will be listed in the location counter field of the listing. If the operand is absolute then the label will be considered absolute. If the operand is relocatable then the label will be considered relocatable. Notice that a label is always required with this statement.

3) DEFL

This statement takes the form

```
     label   DEFL    value operand
```

and instructs the assembler to set the symbolic label specified to the value of the operand specified. This value will be listed in the location counter field of the listing. The difference between this statement and the EQU statement is that the same label may be DEFL'ed many times to different values in the same program while an EQU'ated label may only be set once. This statement has been included in M-ZAL to allow compatibility with other TRS-80 assemblers although from a software engineering viewpoint its use is discouraged.

Only the last value of a DEFL label will appear in the symbol table listing. If a DEFL label is used as an ENTRY then it will be known externally only by its last value.

4) END

This statement takes the form

label END value operand

and denotes the end of the source program. Any statements that appear after an END statement will be ignored. Both the label and the operand are optional on this statement. The operand is used to specify the transfer address of the object module just assembled. If omitted, the transfer address is taken to be the address of the first byte of object code that the source program generated. The transfer address will be listed in the location counter field of the listing. Note that the LINKER always considers a module's transfer address to be relocatable. The LINKER can also be used to manually modify a module's transfer address.

The use of a label on the END statement is optional. If used, the label will take on the value of the current location counter.

5) DEFS

This statement takes the form

label DEFS value operand

and causes the assembler to increment the location counter by the value of the operand specified. The operand is always considered absolute. The value of the operand will be listed in the location counter field of the listing. If a label is used on this statement, it takes on the value of the current location counter (prior to the incrementing operation).

This pseudo-opcode is commonly used to reserve areas of storage for variables named in the label field. No object code is generated by this pseudo-opcode.

6) BYTES

This statement takes the form

label BYTES value operand1{,value operand2}

and causes the assembler to initialize each byte in a contiguous area of storage to the same value. Value operand1 specifies how many bytes are to be initialized (0 is invalid). Value operand2 specifies the value of each byte. Value operand2 is optional and if omitted its value defaults to 0. Examples:

```
BYTES   50,' '   ;generates 50 bytes of blanks
BYTES   4        ;generates four bytes of zeros
```

Both operands are always considered absolute. Labels which are external (EXTRN) may not be used in either operand. The listing will only show the first byte generated.

7) WORDS

This statement takes the form

```
     label    WORDS    value operand1{,value operand2}
```

and causes the assembler to initialize a contiguous number of words to the same two byte value. Value operand1 specifies how many 2 byte words are to be initialized (0 is invalid). Value operand2 specifies the value of each word. Value operand2 is optional and if omitted its value defaults to 0. Both operands are always considered absolute. Labels which are external (EXTRN) may not be used in either operand. The listing will only show the first word generated.

8) DEFB

This statement takes the form

```
     label    DEFB     value operand
```

and causes the assembler to generate one byte of object code specified by the operand.

9) DEFW

This statement takes the form

```
     label    DEFW     value operand
```

and causes the assembler to generate one word (two bytes) of object code specified by the operand. The word is assembled in the standard Z-80 lsb-msb format (least significant byte - most significant byte). Therefore the statement

```
     START    DEFW     7000H
```

will cause a 0 to be assembled at location START and a 70H to be assembled at location START+1.

The operand of a DEFW statement may be absolute, relocatable, or external (EXTRN).

10) DEFM

This statement takes the form

```
     label    DEFM     'character string'
```

and causes the assembler to generate a string object bytes correpsonding to the ASCII codes for each character specified.

The character string may be as long as will fit on a source line. If the quote character itself (') is desired as part of the string then it should be specified as two consecutive quotes. A special listing format is used when this pseudo-opcode appears on the listing to allow for efficient use of paper.

11) ENTRY

This statement takes the form

```
ENTRY   label1{,label2,label3,...}
```

and causes the assembler to define each label specified as an ENTRY into the program. A label which is defined as ENTRY in one program module can be accessed in other program modules (where it will be defined as EXTRN).

At least one label operand is required on an ENTRY statement although you can specify as many as you wish, just seperate them with commas. If an error is detected while parsing a label then all the rest of the labels on the line will be ignored. Each label defined as ENTRY must have a value assigned to it somewhere else in the program (i.e.: by an EQU statement, use on a object code generating statement, etc).

The value of each ENTRY label and its attributes (absolute or relocatable) are placed in the /RLD file and are available to the user at "link" time (when the LINKER program is used).

Because the ENTRY statement generates no object code, its label field must be blank.

12) EXTRN

This statement takes the form

```
EXTRN   label1{,label2,label3,...}
```

and causes the assembler to define each label specified as external (EXTRN) to the program. Such labels must not be defined anywhere else in the program, they can only be referenced. Labels are defined as EXTRN because their value is not known at assembly time. All references in the program to EXTRN labels must be two byte operand references. These references will be assembled as zero, but it will be possible to modify them at "link" time.

Each EXTRN label, and all of the references to it, are placed in the /RLD file and thus passed to the LINKER program. The references will be automatically resolved by the LINKER when it merges this program module with another one which defines the same name as ENTRY. It is also possible to manually set the value of an EXTRN label and resolve its references.

At least one label operand is required on an EXTRN statement although you can specify as many as you wish, just seperate them with commas. If an error is detected while parsing a label then all the rest of the labels on the line will be ignored. Because the EXTRN statement generates no object code, its label field must be blank.

VII. ASSEMBLER COMMAND STATEMENTS

Assembler command statements are used to control the actions of the assembler during the assembly. They do not generate or affect the generation of object code. All assembler command statements begin with a * in column 1. The command verb follows the * immediately with no intervening blanks. The available command verbs are

LIST EJECT SPACE TITLE PAUSE INCL

1) *LIST OFF

If you intend to request a listing at assembly time, but do not wish to have the entire program listed, this command can be used. This command causes the assembler to stop producing a listing until a *LIST ON command is encountered. While *LIST OFF is in effect the only statements that will be listed are those that have errors. Furthermore, any *EJECT, *SPACE, and *PAUSE statements that are encountered will be ignored.

2) *LIST ON

This command negates the effect of the *LIST OFF command just discussed. *LIST OFF and *LIST ON commands cannot be nested. The state of the listing is always determined by the last *LIST statement encountered.

3) *EJECT

If a listing is being generated on the printer then this command will cause the printer to advance to the top of the next page (unless it is already at the top of a page). The *EJECT statement itself is not listed.

4) *SPACE {value operand}

If a listing is being generated on the printer then this command causes the printer to advance by the number of lines specified in the operand. If the operand is omitted then a value of 1 is assumed. This command will not cause the printer to advance beyond the top of a new page. The *SPACE statement itself is not listed.

5) *TITLE 'character string'

If a listing is being generated on the printer then this command causes the printer to advance to the top of the next page and print a title line consisting of the character string specified. Each subsequent page will also be given this title line (until such time as it is changed by another *TITLE command). The maximum length of the title character string is 64 characters. The quote character itself may be embedded in the title character string by specifying it as two consecutive quotes. If a zero length character string is specified ('') then all subsequent title lines will be blank. The *TITLE statement itself is not listed.

6) *PAUSE

If this command is encountered while a listing is being generated (to display and/or printer) and the "Wait on Pause" option was selected then it will cause the assembler to pause. The user can resume the assembly by hitting the ENTER key or terminate it by hitting the CLEAR key. The *PAUSE statement itself is listed just before the assembler pauses. All *PAUSE statements in a program can be effectively ignored by turning off the "Wait on Pause" option before starting an assembly.

7) *INCL program module name

This command is a very powerful feature of M-ZAL. Even though the largest source file that may be edited by TXEDIT is determined by the amount of memory in your system, the largest source <u>program</u> that may be assembled by ASMBLR is actually limited only by the amount of online disk storage in your system. This is possible because of the *INCL command, which stands for INCLude.

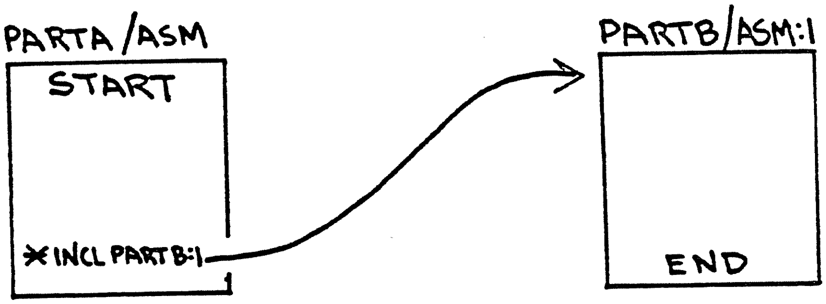
When the assembler encounters a *INCL statement it opens the source (/ASM) file specified by the statement's operand and begins taking source statements from the file. When the end of the file is reached, the assembler continues with the statements following the *INCL statement. This process can be nested to a maximum depth of 4. It is therefore possible to chain together several disk source files into one assembler source program.

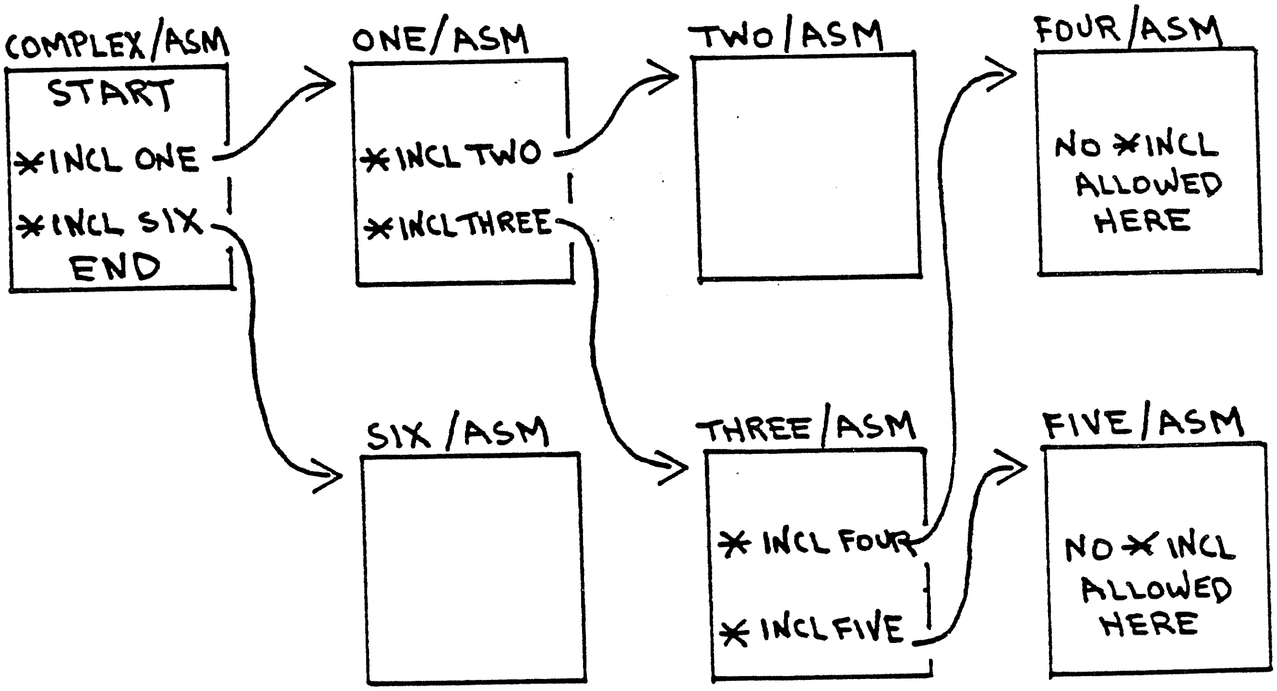
Note that the *INCL statement operand is specified in the form of a module name and that the assembler automatically adds the /ASM extension to the name specified. If necessary, a disk drive qualifier can be added to the module name. For example, the statement *INCL BETA:3 will cause the assembler to open the disk file named BETA/ASM:3 .

If you attempt to nest *INCL statements more than 4 deep, you will get an "INVALID OPCODE" error, this is shown in the diagram on the following page.

The sample program DEMO02 provided with the M-ZAL system contains an example of the *INCL statement. The statement is used to include a source file named ROMEQU/ASM which is also provided on the M-ZAL diskette. This file contains no object code. It is a list of EQUates that define the ROM addresses of useful TRS-80 routines. You may find it useful to *INCLude this file in your own programs.

Examples of *INCL usage:





VIII. THE ASSEMBLER LISTING

As mentioned earlier an assembler listing can be directed to the video display and/or printer. When a listing is sent to the printer it is formatted into pages. Each printed page is numbered and given a title line (see *TITLE command). The printer paper should be setup prior to starting an assembly so that the first line will be printed at the top of a page. The assembler assumes that each printed page will contain 66 lines. Therefore you should set up your printer to print at 6 lines per inch if you are using 11 inch forms, or 8 lines per inch if you are using 8½ inch forms. (If you use roll paper you can ignore this.)

The assembler listing contains three fields to the left of each source line. The leftmost field is the location counter field. Whenever object code is generated this field contains the current value of the location counter (in hexadecimal). Certain pseudo-opcodes cause other values to be listed in this field.

The next field is the object code field and contains any object code generated by the source line. This field is also always listed in hexadecimal.

The next field is the line number field and contains the line number of the current source line in decimal. Line numbers are generated when the source file is created/edited by TXEDIT.

Following the line number field the source statement itself appears. The assembler always assumes a maximum printer width of 80 characters. If a source statement is too long to fit on the current print line then it will be continued in column 1 of the next print line.

If you use DOSPLUS instead of TRSDOS please note: DOSPLUS will interfere with M-ZAL hardcopy formatting unless the DOSPLUS command "FORMS (P=66,L=66)" is issued before invoking ASMBLR.

IX. THE SYMBOL TABLE LISTING

The symbol table listing and/or cross-reference (XREF) can be obtained by selecting the appropriate options on the full screen option menu. If no "List to" option is selected then the symbol table listing will appear on the video display. If the listing is going to the printer then an EJECT will be performed and the title will be set to correspond to the field headings of the symbol table. If no labels appear in the assembler source then the request for a symbol table listing will be ignored.

The following is a description of each field provided in the symbol table listing, from left to right.

Statement of Definition	- The line number (in decimal) of the source statement in which the label was defined.
Value	- The hexadecimal value of the label.
Attributes	- The attributes of the label can be any combination of the following:

R - relocatable	E - ENTRY	D - DEFL
A - absolute	X - EXTRN	

Label	- The label itself.
XREF (optional)	- A list of all the line numbers (in decimal) that the label was referenced in. The format is set up for an 80 column printer.

X. ERROR MESSAGES

At the end of an assembly the total number of error messages produced will be displayed. The possible causes of each error message are discussed in this chapter. Error messages always <u>precede</u> the statement that caused them in the listing. It is possible for one statement to generate more than one error message.

1) INVALID LABEL

- The characters in the label field of an assembler language statement are invalid or form an invalid label. The label field will be ignored and the rest of the line will be handled normally.
- A valid label appears in the label field but the pseudo-opcode specified does not allow labels. The label will be ignored and the rest of the line will be handled normally.
- No label appears in the label field but the pseudo-opcode specified requires one.

2) NO OPCODE

The current assembler source statement is missing the opcode field. A blank source line without a semicolon (;) to mark it as a comment will produce this error.

3) INVALID OPCODE

The character string found in the opcode field does not form a valid Z-80 opcode, M-ZAL assembler pseudo-opcode, or M-ZAL assembler command verb. The entire source line is ignored. This message is also given if an attempt is made to nest *INCL statements more than 4 deep.

4) INVALID OPERAND(S)

At least one operand parsed on the current line was not a valid operand or not a valid operand for its opcode. This error will supercede other errors on the same line and causes the line to be ignored in most cases.

5) UNDEFINED LABEL(S)

- An EXTRN label has been used in an expression.
- A label used in a value operand is undefined. Object code will be generated using an operand value of 0.

Also note the following:

- A label equated to an undefined value operand will remain undefined. A label DEFLed to an undefined value operand will retain its previous value.
- An undefined label will not appear in the symbol table cross reference listing.
- An undefined label which has been specified as ENTRY will not be placed in the /RLD file.

6) MULTIPLE DEFINED LABEL

A label that has already been defined has been defined again or has been given conflicting attributes. The new value or attribute will be ignored. (A remote possibility is that an ORG, DEFS, BYTES, or WORDS statement depends upon itself; this is explained in the next chapter.)

7) EXPRESSION OUT OF RANGE

A value operand is too big to fit into the field it was specified for. The overflow or high-order bits are ignored when code is generated. If this occurs with the IM or RST opcodes a value operand of zero will be used to generate code.

8) JR OUT OF RANGE

A relative jump (JR) to a address too far away has been requested. The current location counter will be used as the jump target address instead.

9) & OR < OPERATOR ON RELOCATABLE LABEL(S)

The cause of this error (which is actually a "warning") is described in section V.3. Statement parsing and code generation continues but caution should be exercised when relocating the resulting object code.

10) INVALID USE OF RELOCATABLE LABELS

The cause of this error is described in section V.3. The value operand will be forced to absolute. A probable error will occur if the resulting object code is relocated. If you are doing this on purpose, exercise caution.

11) NO END STATEMENT

An end of file condition was reached in the assembler source file before an END statement was encountered. The transfer address will be set to the location counter of the first byte of generated object code.

12) OUT OF MEMORY

Certain functions use variable amounts of memory during the assembly. The largest user of memory is the internal symbol table. The "Generate /RLD" and "Symbol table XREF" options also require extra memory.

This error causes immediate termination of the assembly. If you do not require the /RLD file or XREF then try running the assembly again with these options turned off. If the error still occurs, or if you must have these options on, then split the program into two or more parts and link them together using the LINKER. This process is described in detail in the LINKER SECTION of this manual.

13) DOS ERRORS

These errors cause immediate termination of the assembly. The error message specifies the DOS error code in decimal as well as the name of the file on which the error was encountered. Typical error codes are:

15 - write protect on diskette.
24 - file not found.
27 - disk space full.
28 - attempt to read past EOF. This can occur if a file has been accidently truncated or is missing records.

Consult your DOS manual for more details.

XI. PHASES OF ASSEMBLY

This chapter describes what happens internally during an assembly. It is not necessary to read this chapter in order to properly use the assembler.

Each assembly consists of 3 phases. The first phase consists of one or more passes through the source program. The second and third phase each consist of one pass through the source program. The third phase is optional.

1) Phase 1

During the first phase all symbols (labels) are resolved. The symbol table is constructed. On exit from phase 1 all the labels that can be defined have been defined.

The assembler will make as many passes as are needed to define all the labels. For example, the following source program will cause the assembler to make one pass during phase 1 :

```
        ORG     7000H
START   LD      HL,0     ;defines START
LABEL   EQU     START    ;defines LABEL
        END
```

However, the following program will cause the assembler to make two passes during phase 1 :

```
        ORG     7000H
LABEL   EQU     START    ;needs definition of START
START   LD      HL,0     ;defines START
        END
```

From the above it can be seen that EQU statements can be placed in any order in the program and it will assemble without errors. Many assemblers make only one pass during phase 1 and would therefore be unable to assemble the 2nd program shown above. The M-ZAL assembler will make as many passes as are necessary to resolve such situations. Regardless of this feature, the user should try to order EQU statements so as to minimize the number of passes required. This will insure a reasonable assembly speed and will reduce disk drive wear.

A very uncommon kind of error can occur because of the multiple pass feature. This error occurs when an ORG, DEFS, BYTES, or WORDS statement depends upon a label that depends upon the statement. This recursive definition conflict will cause all the symbols in the affected portion of the program to be multiply defined. Unless you purposely generate this error you should never come across it. The following is an example of a source program that contains the recursive definition error:

```
        ORG     LABEL   ;loc counter depends on LABEL
START   LD      A,0
        LD      HL,0
LABEL   LD      A,B     ;LABEL depends upon ORG which
;                        depends upon LABEL which....
LAB1    LD      HL,0
        END
```

In the above program the value of the ORG operand depends upon the value of the location counter, but the ORG statement must set the location counter! In this program all three labels START, LABEL, and LAB1 will have MULTIPLE DEFINED LABEL errors.

2) Phase 2

Phase 2 consists of one pass through the source program. The assembler listing, the /CMD file, and the /RLD file are all generated during this phase. Even if none of these options have been selected, this phase always takes place so that any errors that exist can be displayed.

3) Phase 3

Phase 3 consists of one pass through the source program and only takes place if "Object to Tape" or "Symbol table XREF" have been requested. If a symbol table listing or XREF was requested then it will be produced at the end of phase 3.

LINKER SECTION

I. THE PURPOSE OF THE LINKER

Just as the assembler is a program which processes disk files containing source text, the linker is a program which processes disk files containing object code. In particular, the linker is used to create modified versions of object code programs, and to combine several object code programs ("modules") together to form larger "composite" programs.

Use the linker whenever you want to relocate a program without having to re-assemble it. To do this you must assemble the program once and create an object code file (name/CMD) as well as a relocation and external symbol file (name/RLD). Once this is done you can instruct the linker to read in the two files, relocate the object code to a different address area, and then write out a new object code file.

Suppose you have a program which makes reference to some value that you would like to parameterize. For example, a program designed to run on both a Model 1 and Model 3 machine might need to make reference to the memory location where DOS stores the highest memory address. This location is different in Model 1 and Model 3 systems.

A conventional approach to this problem is to reference the location symbolically as TOPMEM and use an EQUate statement at the beginning of the program to set its value. The user must then assemble the program to create the object code for one machine configuration, edit the source program to change the TOPMEM EQUate for the alternate configuration, and then re-assemble the program.

The M-ZAL approach to this problem would be to remove the TOPMEM EQUate from the program altogether and instead define TOPMEM as EXTRN (external). The program is then assembled and object code and relocation/external symbol information is produced. Since it is an external symbol, the linker can now be used to manually modify the value of TOPMEM. The user can therefore create several different copies of the object code program, each with a different value for TOPMEM.

This approach yields the same results in much less time than it would take to perform multiple assemblies. In addition, it makes it possible to modify the program in the future without requiring that the source be kept around. This can be a major consideration when it is not convenient to maintain the source program online.

The linker's most powerful function, however, is the combining of several independently assembled program modules. In this application, linkage between the modules is accomplished by the linker through the resolution of EXTRN and ENTRY symbol names.

This facility gives you the ability to make use of general purpose subroutines without having to include the source code for the subroutines in your main program. You can create a set of such subroutines, assemble them once, and then re-use them over and over again in many different applications. It is not even necessary to keep the source code for such routines, as long as you have a concise description of how to call them.

The M-ZAL diskette includes several routines of this nature. These will be used in the examples that follow and you may even find them to be of general use in your own programs.

If you have large and complex programming applications you can break them down into program modules which are smaller and more manageable and then use the linker to "put the pieces together" and create the final product. It is even possible to have a different programmer working on each module. Since the modules are not assembled together, there is no problem if two of them end up using the same symbolic name for different purposes. At the begining of such a development effort the function of each module is clearly defined and the names of all of the global symbols (ENTRYs and EXTRNs) are specified.

II. INVOKING THE LINKER

The linker is invoked from DOS READY mode by simply typing in the command LINKER. The linker will initialize and display its prompt character which is the plus sign (+). You control the linker by typing in linker commands whenever the prompt character is present. The linker commands are:

```
BUILD      END      EXEC      INIT      LOADA      LOADR      MAP
ORG        SET      TAPE      TRAN      XREF
```

Whenever the linker completes the execution of any of these commands it always displays the following message:

```
        ORG=xxxxH        TRAN=xxxxH        UNRESOLVED=nn
```

These are the three primary linker parameters which you must monitor as you use the linker to build new object files. The ORG parameter is the next address to which the linker will relocate a module. The TRAN parameter is the current transfer address (main entry point) for the composite object program being built. The UNRESOLVED parameter is the current (decimal) number of unresolved symbols. An unresolved symbol is one which appears as EXTRN in some module which has been loaded but does not yet appear in any module as ENTRY.

III. LINKER COMMANDS

In this chapter each of the linker commands will be described in detail. After reading this chapter you should proceed directly to the next chapter in which a sample application of these commands will be demonstrated.

1) Specify Origin - ORG

The ORG command is used to set the linker's ORG value. The ORG value represents the next address to which the linker will relocate a program. When the linker is initialized the ORG value is set to 7000H which is the standard starting address for disk-based machine language programs.

To use the ORG command type in ORG followed by one or more blanks and then the value that you wish to set the ORG to. The value should be specified as a numeric constant using the standard assembler language syntax. For example, the command ORG 7024H would set the linker's ORG value to 7024H. The command ORG 28672 would set the linker's ORG value to 7000H (28672D = 7000H).

2) Load and Relocate - LOADR

The LOADR command is used to instruct the linker to load in a (object code) program module and relocate it to the current linker ORG value. When this command is completed the linker's ORG value is set to the next address above the module just loaded and the linker's TRAN value is set to the transfer address of the module just loaded.

To use this command type in LOADR followed by one or more blanks and then the name of the program module that you wish to load. For example, the command LOADR ALPHA would cause the linker to load in the object code contained in file ALPHA/CMD and relocate it to the current ORG value by using the relocation information in file ALPHA/RLD. Any global symbols (ENTRYs and EXTRNs) in module ALPHA will be added to the linker's symbol table.

If desired, a disk drive designator can be added to the module name. For example, the command LOADR ALPHA:1 would cause the linker to refer specifically to the files named ALPHA/CMD:1 and ALPHA/RLD:1.

3) Load Absolute - LOADA

This command functions in a manner very similar to the LOADR command just described. However, the object code is loaded into memory at whatever address was specified when it was assembled, no relocation takes place. The linker's ORG value is not changed by this command. The linker's TRAN value is set to the transfer address of the module just loaded. Since there is no relocation, the module's /RLD file need not be available when this command is used. Of course, if it is not present then no global symbol (ENTRY and EXTRN) information will be associated with the module.

4) Display Module Map - MAP

The MAP command causes the linker to display a map of all modules currently loaded. The map is sorted by the starting address of each module. The address ranges occupied by each module are provided on the map, as well as each symbol which is defined in the module as either ENTRY or EXTRN. If a symbol is unresolved then it is prefixed by an "*" in the listing.

During the execution of this command the linker may be paused by holding down the SPACE BAR. Once paused, it is necessary to hit the ENTER key to proceed.

The map can be routed to the printer instead of to the display screen by adding the operand "P" to the command, thus:

```
MAP   P
```

5) Display Symbol Table Cross-Reference - XREF

The XREF command causes the linker to display an alphabetically sorted list of all currently defined symbols. Each symbol is followed by a list of the modules in which it is EXTRN. If the symbol has been resolved then its address value is also shown, as well as the name of the module that it is ENTRY to. Otherwise, the symbol's address value appears as ????.

If the symbol has been assigned a value by the user via a SET command (to be described shortly, hang on) then this is indicated by the words "*USER*SET*" in the ENTRY module name field.

During the execution of this command the linker may be paused by holding down the SPACE BAR. Once paused, it is necessary to hit the ENTER key to proceed.

The xref can be routed to the printer instead of to the display screen by adding the operand "P" to the command, thus:

```
XREF   P
```

6) Specify Transfer Address - TRAN

As mentioned earlier, the linker's TRAN value is the main entry point (transfer address) that the linker will specify when instructed to create a composite object code file (via the BUILD or TAPE commands).

Normally this value is equal to the assembled transfer address of the last module loaded (via the LOADA or LOADR commands). The TRAN command gives you the ability to set the linker's TRAN value manually. There are two ways to do this. One way is to use the TRAN command with a numeric constant in the standard assembler language syntax. For example, the command TRAN 402DH will cause the linker's TRAN value to be set to 402DH.

The alternative approach is to specify the TRAN value as the value of a symbol which is known to the linker. For example, if you have loaded in a module named BETA which contains an ENTRY symbol named BETA5 then the command TRAN BETA5 will cause the linker's TRAN value to be set to the address value of symbol BETA5 within module BETA. The symbol must be defined and resolved at the time this command is issued or an error message will be given.

7) Create Composite /CMD File - BUILD

This command causes the linker to resolve all external references in all modules that have been loaded. If references exist to a symbol that has not been resolved (symbols are resolved either manually by the SET command or by loading in a module that specifies them as ENTRY) then they are ignored.

The linker then creates a composite /CMD file using the name specified in this BUILD command. This composite will contain all of the object code from all of the modules that have been loaded and will specify a transfer address equal to the linker's current TRAN value. Examples:

The command BUILD CHARLIE will create a composite object code file named CHARLIE/CMD.

The command BUILD CHARLIE:1 will create a composite object code file named CHARLIE/CMD:1.

8) Create Composite SYSTEM Tape - TAPE

This command performs the same function as the BUILD command just discussed except that the composite object code is written out to tape in the standard Radio Shack "SYSTEM tape" format. For example, the command TAPE TEST3 will create a SYSTEM tape file named TEST3. Make sure that your cassette machine is setup to "record" prior to entering this command.

Note for Model 3 only: the baud rate (speed) at which tape operations take place is determined by the contents of a special memory address. Refer to your Radio Shack Model 3 manual for details.

9) Set Symbol's Address Value - SET

This command allows the user to manually set the address value for a symbol. For example, the command SET BETA5 01C9H will cause the value of symbol BETA5 to be set to 01C9H.

The symbol specified must be known to the linker (i.e.: be defined as ENTRY or EXTRN in some module) before this command is entered or an error message will be given. If the symbol specified was unresolved prior to entering this command then this action will cause the symbol to become resolved.

On the other hand, if the symbol specified was previously defined as an ENTRY to some module then the association between the symbol and the module will be broken (i.e.: the symbol will no longer be considered an ENTRY to the module and will no longer appear under the module in a MAP listing).

Any symbol which has been set by this command will be so indicated in the symbol table XREF listing.

10) Initialize Linker - INIT

This command is used to re-initialize the linker. No operands are required. All LOADed modules are cleared from memory and the linker's ORG value is reset to 7000H. The linker's TRAN value is reset to 0 and the symbol table is emptied.

This command makes it possible to build several different composite programs with one invocation of the linker. It also comes in handy when you discover that you accidently loaded a module into the wrong memory area and need to "start over."

11) Terminate Linker - END

This command (which requires no operands) causes the linker to terminate and return to DOS READY mode.

12) Execute Control File - EXEC

This command causes the linker to start executing commands from the CTL file specified. For example, the command EXEC COMNDS:1 will cause the linker to read and execute commands from the file named COMNDS/CTL:1.

The CTL file is any standard text file created by the text editor program (TXEDIT). It may contain any valid linker commands except another EXEC command. After the last command in the CTL file is executed, control returns to the keyboard.

If an error occurs while executing commands from a CTL file the linker will pause. At such a time you must either hit the ENTER key to continue processing the CTL file, or hit the CLEAR key to return control to the keyboard.

Note that the linker will ignore any command line that begins with a semi-colon (";"). This makes it possible to add documentation (comment lines) to CTL files.

This command is useful when you are working on a large program which consists of several modules. Suppose you have all of the modules working except one. You have to keep assembling this last module until you get it working. Each time it is re-assembled you will probably have to go thru a lengthy linker session to construct the composite /CMD file for testing. In such a situation you could use the text editor to create a /CTL file containing the required sequence of linker commands. Once this is done you need only enter the EXEC command specifying this /CTL file and the linker will do the rest.

IV. SAMPLE LINKER SESSION

In this chapter we will demonstrate the usage of the linker in combining a main program with several external (independently assembled) subroutines. The main program is the DEMO02 program which is included on your M-ZAL diskette in source format (DEMO02 /ASM). You may wish to use the text editor to browse through this source file right now.

The DEMO02 program is a marginally useful program which turns your TRS-80 into a hexadecimal - decimal conversion calculator. It makes use of the general purpose conversion subroutines which are also provided on your M-ZAL diskette. These subroutines are packaged into two seperate program modules:

DECCONV - decimal conversion routines
HEXCONV - hexadecimal conversion routines

and are described in detail in the Appendix.

We start by assembling the main program, DEMO02. Use ASMBLR and be sure the OBJECT TO DISK and CREATE /RLD options are selected. The main module, DEMO02, is now available. The program is not usable, however, because it references external routines which have not yet been linked to it.

Now invoke the linker. Note that ORG is set to 7000H which is just fine for our purposes. Note also that there are no UNRESOLVED symbols at this time, as there are no symbols at all! Now type in the command `LOADR DEMO02`

The linker will load in the DEMO02 module and display the message:

```
ORG = 71B4H     TRAN = 7000H     UNRESOLVED = 04
```

Note that the ORG has been updated to the first free address after the DEMO02 module just loaded, 71B4H. The TRAN value has been set to the transfer address of the DEMO02 program, 7000H. Note also that there are now four unresolved symbols. These are, of course, the external references of the DEMO02 program. Now type in the command MAP. The linker will display:

```
                          MODULE MAP
                          ------ ---

MODULE NAME  ADDRESSES  ENTRIES            EXTRNS
-----------  ---------  -----------------  -----------------
DEMO02       7000-71B3                     *ADCONV   *AH4CONV
                                           *DACONV   *HA4CONV
-----------  ---------  -----------------  -----------------
```

This indicates the addresses occupied by the DEMO02 module, as well as the external (EXTRN) symbols in the DEMO02 module. Note that these symbols are as yet unresolved, as indicated by the * which prefaces them. Now type in the command XREF. The linker will display the following:

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

 SYMBOL    ADDR   ENTRY IN     EXTRN IN
--------   ----   ----------   ------------------------------------
ADCONV     ????   UNRESOLVED   DEMO02
AH4CONV    ????   UNRESOLVED   DEMO02
DACONV     ????   UNRESOLVED   DEMO02
HA4CONV    ????   UNRESOLVED   DEMO02
--------   ----   ----------   ------------------------------------
```

This is an alphabetical listing of all symbols known to the linker. At present the only symbols known to the linker are those that were defined as EXTRN in module DEMO02. They are as yet unresolved. Now type in the command LOADR DECCONV.

The linker will load in module DECCONV and display the following:

```
      ORG = 7241H     TRAN = 71B4H      UNRESOLVED = 02
```

The first item to notice is that the number of unresolved symbols has dropped from 4 to 2. The DECCONV module contained as ENTRYs two of the symbols that were EXTRN in DEMO02. These symbols are now resolved. Now type in the command MAP. The linker will display:

```
                          MODULE MAP
                          ------ ---

MODULE NAME  ADDRESSES  ENTRIES            EXTRNS
-----------  ---------  -----------------  -----------------
DEMO02       7000-71B3                      ADCONV   *AH4CONV
                                            DACONV   *HA4CONV

DECCONV      71B4-7240  ADCONV    DACONV
-----------  ---------  -----------------  -----------------
```

Notice that module DECCONV has been loaded into memory after module DEMO02, the linker's ORG value is now set to the first address after module DECCONV, and the linker's TRAN value is now set to the transfer address of the DECCONV module. The two symbols ADCONV and DACONV are now resolved as ENTRYs to module DECCONV. To obtain the actual address values for these symbols type in the command XREF. The linker will display:

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

 SYMBOL     ADDR   ENTRY IN     EXTRN IN
--------    ----  ----------   ---------------------------------
ADCONV      71F6  DECCONV      DEMO02
AH4CONV     ????  UNRESOLVED   DEMO02
DACONV      71B4  DECCONV      DEMO02
HA4CONV     ????  UNRESOLVED   DEMO02
--------    ----  ----------   ---------------------------------
```

Now type in the command LOADR HEXCONV . The linker will load in module HEXCONV and display the following:

```
      ORG = 7292H      TRAN = 7241H      UNRESOLVED = 00
```

The remaining unresolved symbols have now been resolved. Note that the ORG and TRAN values have been updated in the usual manner. Now type in the command MAP. The linker will display the following:

```
                              MODULE MAP
                              ------ ---

MODULE NAME  ADDRESSES  ENTRIES                 EXTRNS
-----------  ---------  -----------------       -----------------
DEMO02       7000-71B3                           ADCONV    AH4CONV
                                                 DACONV    HA4CONV

DECCONV      71B4-7240  ADCONV    DACONV
HEXCONV      7241-7291  AH2CONV   AH4CONV
                        HA2CONV   HA4CONV
-----------  ---------  -----------------       -----------------
```

This is now a complete map of the composite object program containing all required modules. Notice that the HEXCONV module contained two additional ENTRYs that are not referenced. To get a complete list of all symbols now known to the linker type in the command XREF. The linker will display:

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

 SYMBOL     ADDR   ENTRY IN     EXTRN IN
--------    ----  ----------   ---------------------------------
ADCONV      71F6  DECCONV      DEMO02
AH2CONV     726C  HEXCONV
AH4CONV     7262  HEXCONV      DEMO02
DACONV      71B4  DECCONV      DEMO02
HA2CONV     7246  HEXCONV
HA4CONV     7241  HEXCONV      DEMO02
--------    ----  ----------   ---------------------------------
```

Now that we have loaded all required modules we are just about ready to instruct the linker to create (BUILD) the composite object code file. However, since the last module loaded was HEXCONV, the linker's TRAN value is set to its transfer address.

We want the transfer address for the composite program to be the main entry point in the DEMO02 program so type in the command TRAN 7000H (remember that when DEMO02 was loaded the TRAN value was set to 7000H). An alternate approach would have been to load the subroutines first and the main program last, thus eliminating the need to manually set the TRAN value.

Now type in the command BUILD DEMOCOMP and the linker will create a new file named DEMOCOMP/CMD containing all of the object modules loaded. The complete DEMO02 program can now be loaded and run from DOS READY mode by simply typing in the command DEMOCOMP.

This entire example can be repeated automatically as a demonstration of the linker's EXEC command by typing in the linker command EXEC EXAMPLE . The M-ZAL diskette includes a file named EXAMPLE/CTL which includes all of the linker control commands just used.

V. MULTI-EXTENT OBJECT MODULES

The linker fully supports object modules that use more than one address range. Such modules are called "multi-extent" object modules and usually result when the assembler's DEFS statement is used. For example, the assembler language code

```
7000                           ORG      7000H
7000   210870                  LD       HL,(PARAM1)
7003   C30C70                  JP       CONTINUE
7006             PARAM0   DEFS     2
7008             PARAM1   DEFS     2
700A             PARAM2   DEFS     2
700C   110C00    CONTINUE LD       DE,12
700F   B7                      OR       A
7010   ED52                    SBC      HL,DE
```

will generate object code for address ranges 7000-7005 as well as 700C-7011. No code will be generated where space is reserved by the DEFS statements. If this code were part of a module named MULTIX which was loaded by the linker, the "map" would look like this:

```
                         MODULE MAP
                         ------ ---

MODULE NAME   ADDRESSES  ENTRIES             EXTRNS
-----------   ---------  ------------------  -----------------

MULTIX        7000-7005
              700C-7011
-----------   ---------  ------------------  -----------------
```

As you can see, the linker keeps track of each seperate range of addresses (called "extents") for each module. A problem can arise, however, if you place DEFS statements at the very beginning or end of a program module. Consider the same example with a slight re-arrangement:

```
7000                        ORG     7000H
7000  210B70                LD      HL,(PARAM1)
                  ;         JP      CONTINUE
7003  110C00      CONTINUE LD       DE,12
7006  B7                    OR      A
7007  ED52                  SBC     HL,DE
7009              PARAM0    DEFS    2
700B              PARAM1    DEFS    2
700D              PARAM2    DEFS    2
```

The "map" nows looks like this:

```
                              MODULE MAP
                              ------ ---
MODULE NAME  ADDRESSES  ENTRIES                  EXTRNS
----------   ---------  ------------------       ------------------
MULTIX       7000-7008
----------   ---------  ------------------       ------------------
```

Notice that the linker is not aware of the fact that MULTIX is using addresses 7009-700E. If you now load another module into the linker it will be placed directly after MULTIX and will overlay the addresses used by PARAM0, PARAM1, and PARAM2.

One way to avoid this kind of conflict is to manually keep track of data areas defined with DEFS . A much better way is to not use the DEFS statement altogether. The BYTES and WORDS pseudo-ops were implemented in M-ZAL to serve as an alternative to the DEFS pseudo-op. Because BYTES and WORDS generate object code, the problem just described does not occur with their use.

VI. ERROR MESSAGES

The following is a list of linker error messages, with probable causes for each:

INVALID COMMAND

This message is given for invalid command verbs, missing operands, and invalid numeric constants.

MEMORY FULL

This message appears during the processing of a LOADA or LOADR command if there is not enough memory to store both the object code and the linker's symbol table.

SYMBOL IS UNDEFINED

This message will be given if the symbol specified in a TRAN command is undefined or unresolved. It is also given if the symbol specified in a SET command is undefined.

MODULE ALREADY LOADED

This message will be issued if you attempt to load the same module twice. You probably forgot an INIT command.

BAD RLD FILE

This error can arise during the execution of a LOADA or LOADR command, or during external symbol resolution while processing a BUILD or TAPE command. It indicates that the contents of a /RLD file are not consistent with the contents of its corresponding /CMD file. This can occur if you assemble a module and generate both /CMD and /RLD files, change the source code, and then re-assemble without regenerating a new /RLD file.

DOS ERROR xxH ON cccccccccc

This message indicates that an i/o error has occurred on the file specified (cccccccccc). The DOS error code is given in hexadecimal (xx). Typical codes are:

0FH - write protect on diskette.
18H - file not found.
1BH - disk space full.
1CH - attempt to read past EOF. This can occur if a file has been accidently truncated or is missing records.

Consult your DOS manual for more details.

VII. LINKER COMMAND SUMMARY

All of the linker commands are summarized here, in alphabetical order:

BUILD	mmmmmmmmmmm	Build composite command file named mmmmmmmmmmm.
END	(no operands)	Return control to DOS.
EXEC	mmmmmmmmmmm	Execute commands from CTL file named mmmmmmmmmmm.
INIT	(no operands)	Initialize.
LOADA	mmmmmmmmmmm	LOAD Absolute (no relocation performed) module named mmmmmmmmmmm.
LOADR	mmmmmmmmmmm	LOAD and Relocate module named mmmmmmmmmmm.
MAP	{P}	Display module map. P = on printer.
ORG	####	Set linker ORG to value ####.
SET	cccccccc ####	Set symbol cccccccc to value ####.
TAPE	ssssss	Build composite SYSTEM tape named ssssss.
TRAN	####	Set linker TRAN to value ####.
TRAN	cccccccc	Set linker TRAN to value of symbol cccccccc.
XREF	{P}	Display symbol table cross-reference. P = on printer.

KEY

####	represents a numeric constant.
ssssss	represents a tape file name (6 chars max).
cccccccc	represents a symbol name (8 chars max).
mmmmmmmmmmm	represents a DOS filespec in "module" name format. This is one to eight characters followed by an optional colon and disk drive designator. The LINKER will automatically add the appropriate file suffix depending on the operation. (i.e.: EXEC LINKFILE:2 references the file named LINKFILE/CTL:2 ; BUILD OMEGA creates a file named OMEGA/CMD).

APPENDICES

A. M-ZAL MEMORY USAGE

The ASMBLR and LINKER programs use memory from address 6000H up to the top of RAM as defined to DOS. The "top of RAM" is stored in a special location by DOS. This location is usually referred to symbolically as TOPMEM and is different for Model I and Model III systems.

Model I	TOPMEM	=	4049H
Model III	TOPMEM	=	4411H

When you start DOS it automatically determines the highest address in your system and stores this value at TOPMEM. For example, if you have a 32K Model I system (highest address = BFFFH) then when you start DOS it will store FFBF at addresses 4049 and 404A (note lsb first, then msb).

You can prevent ASMBLR and LINKER from using memory that you wish to reserve at the "top" of RAM by modifying the contents of TOPMEM before invoking them. For example, if you have a 48K Model III system (highest address = FFFFH) and you wish to reserve the top 1K for a special printer driver routine then you would store FFFB at addresses 4411 and 4412 before invoking ASMBLR or LINKER.

* * *

As shipped, the TXEDIT program occupies addresses AB00H - BDFFH. The TXEDIT text buffer is stored in memory starting at 5CF0H. The text buffer can grow until it reaches the start of the editor itself (AB00H). The editor thus allows a maximum text buffer in a 32K system while reserving the top .5K of memory (from BE00 to BFFF) for the user.

If you wish to reserve more "top of RAM" memory at the expense of a smaller text buffer, or if you have a 48K system and wish to make use of the additional 16K then you can do so by relocating the TXEDIT program. This is described in the next section.

B. CUSTOMIZING TXEDIT

There are two reasons for customizing the TXEDIT program. One of these is to allow it to use more or less memory for the text buffer, this was explained in the preceding section. The second reason is to modify the rate at which the auto-repeating control keys repeat. The first change is accomplished by relocating the TXEDIT program, the second by changing the value of two variables within the program. Both customizing changes can be made at the same time; they are made by passing the TXEDIT program through the linker.

If you decide to customize TXEDIT then perform the following step <u>only once</u>. <u>Do not</u> perform this step each time you re-customize.

Rename the copy of the TXEDIT program that was shipped on the M-ZAL diskette by entering the following DOS command:

```
        RENAME  TXEDIT/CMD TO TXBASE/CMD
```

The M-ZAL diskette now contains a matching pair of files named TXBASE/CMD and TXBASE/RLD. Whenever you wish to create a new version of TXEDIT you will do so by instructing the linker to load in module "TXBASE". You will then instruct the linker to perform the desired customization and create (BUILD) a new file named TXEDIT/CMD. For example, if you have a 48K system and wish to make use of all available memory for the text buffer you would enter the following commands to the linker:

```
        INIT
        ORG     0ED00H
        LOADR   TXBASE
```

At this point you would notice that there are two unresolved symbols. A MAP or XREF command would show that these symbols are named DELAY1 and DELAY2. These external symbols were defined to allow you to change the control key auto-repeat rate. To proceed with the example, assuming you do not wish to change the auto-repeat rate, enter the following linker commands:

```
        SET     DELAY1  3500H
        SET     DELAY2  2500H
        BUILD   TXEDIT
        END
```

The above would create a new copy of the TXEDIT program which makes use of all memory in a 48K system and thus provides the largest text buffer possible.

Now let us assume that you are a very fast typist and wish to increase the speed with which the editor's auto-repeating keys repeat. This is accomplished by lowering the value of the two external variables DELAY1 and DELAY2. (Conversely, you can slow down the auto-repeat rate by increasing the value of these variables.) We shall also assume that you have a 32K system and do not wish to relocate the editor. You would create your new version of the editor with the following linker commands:

```
        INIT
        LOADA   TXBASE
        SET     DELAY1  2E00H
        SET     DELAY2  1E00H
        BUILD   TXEDIT
        END
```

You may want to try different values of DELAY1 and DELAY2 so that you find the most comfortable "action" for your keyboard.

From these examples it should be evident that both auto-repeat rate and relocation customization can be performed at the same time.

IMPORTANT NOTE: When relocating the TXEDIT program, be sure that it always starts on a 256 byte memory address boundary (in other words, the low order byte of the starting address (ORG) must be zero). The editor will not function properly if this condition is not met. For example, if you have a 32K system and you need 640 bytes (280H) for a special driver routine, then you must actually reserve 768 bytes (300H) by relocating TXEDIT so that it occupies AA00H - BCFFH. You will have addresses BD00H - BFFFH available for your routine.

As shipped, the TXEDIT program occupies addresses AB00H - BDFFH. If you have a 32K system, this gives you 512 bytes (200H) free from BE00H - BFFFH.

C. FORMAT OF TEXT FILES

Text files are created by the TXEDIT program and are used as input to the ASMBLR and LINKER programs. The format of these files may be different from the format of text files created and used by other manufacturer's software. The following information is provided so that you can convert M-ZAL format text files to and from text files in other formats.

A text file is comprised of one or more physical disk records, each of which is 256 bytes long. The first two bytes of the first record contain a value (in standard lsb-msb format) which defines the size of the file. Subtract 5CF0H from this value to obtain the total number of data bytes in the file. The data bytes follow immediately and span as many physical records as are needed to contain the text. Each line of text is preceded by three data bytes. The first two bytes contain the binary value (in standard lsb-msb format) of the line's line number. The next byte specifies the number of bytes in the line itself, which follow immediately. Note that the line is stored with blanks compressed out wherever possible. The data character 09H is defined to represent a horizontal tab (which means fill blanks until the next tab position).

After the last byte of the last line of text two bytes containing FFFFH will appear.

Example: The text file containing

```
00010   START   LD      HL,0     ;BEGIN PROGRAM
00020           XOR     A
00030           RET
```

would be stored in a single disk sector as

```
byte
off-              Hexadecimal Data                     Ascii Data
set

00    225D 0A00 1D53 5441 5254 094C 4409 484C   "....START.LD.HL
10    2C30 093B 4245 4749 4E20 5052 4F47 5241   ,0.;BEGIN PROGRA
20    4D20 1400 0709 584F 5209 4120 1E00 0509   M ....XOR.A ....
30    5245 5420 FFFF 0000 0000 0000 0000 0000   RET ............
40    0000 ...
.       ...
```

D. FORMAT OF /RLD FILES

A /RLD file is created by the ASMBLR program when the GENERATE /RLD option is selected. This file contains information that is used by the LINKER program when it loads the corresponding object module. The information is used to perform relocation and external symbol resolution.

The /RLD file contains three kinds of information:

1) A list of location counter values corresponding to each location in the object code where a relocatable value was assembled.
2) A list of symbolic names that were defined as ENTRY in the program, their associated values, and whether or not they are relocatable.
3) A list of symbolic names that were defined as EXTRN in the program. For each symbol so defined, a list of location counter values is also provided. These values correspond to each location in the object code where a reference to the associated EXTRN symbol was made.

A /RLD file is comprised of one or more physical disk records, each of which is 256 bytes long. The first two bytes of the first record contain a value (in standard lsb-msb format) which defines the number of entries in list 1. A value of zero is valid here, and would indicate that there were no relocatable values in the associated object code. The next two bytes contain a value which defines the number of entries in list 2. Once again, a value of zero is valid and indicates that no symbols were defined as ENTRY in the program. The next two bytes contain a value which defines the number of entries in list 3. A value of zero is valid and indicates that no symbols were defined as EXTRN in the program. These bytes are followed immediately by the data for list 1, then the data for list 2, and lastly, the data for list 3.

Each entry in list 1 is a two byte value in standard lsb-msb format. Each entry in list 2 is 10 bytes long. The first 8 bytes contain the ENTRY symbol name, left justified and padded on the right with blanks as needed. The last 2 bytes of each list 2 entry contain the value of the associated ENTRY symbol in standard lsb-msb format. If the ENTRY symbol is relocatable, the high-order bit of the last byte of the symbol name is turned on. Each entry is list 3 is of variable length. The first 8 bytes contain the EXTRN symbol name, left justified and padded on the right with blanks as needed. The next two bytes contain a value which indicates the number of references in the object code to the associated EXTRN symbol. The location counter value for each of these references then follow.

E. FORMAT OF /CMD FILES

A /CMD file is created by the ASMBLR program when the OBJECT TO DISK option is selected. This file contains the generated object code in a format which is loadable by DOS. There are also several DOS commands and utilities (such as the DUMP command) which create /CMD files.

A /CMD file is comprised of one or more physical disk records, each of which is 256 bytes long. It is easier to view this file, however, as containing one or more logical records, each of which is of variable length. Logical records span physical records as needed.

The first byte of each logical record defines the record type. The second byte of each logical record defines the length of the record by specifying how many data bytes follow. The data bytes follow immediately and are interpreted depending on the record type.

Record type 1 (first byte = 01H) contains object code to be loaded. The first two data bytes (following the length byte) contain the address (lsb-msb) to load. This is followed immediately by "n" bytes of object code, where "n"= the record's length byte - 2. The 2 is subtracted to take into account the 2 bytes of address information.

Record type 2 (first byte = 02H) signifies the end of the /CMD file and specifies its transfer address. Its length byte is always 2 and is followed by the 2 byte transfer address (lsb-msb).

Record type 5 (first byte = 05H) is used to add documentation to the /CMD file. The data in these records is ignored when the file is being loaded by DOS. The amount of data contained in such records is variable and is specified, in the usual fashion, by the second byte of the record (the length byte).

No other record types are known to be defined for /CMD files, though it is generally believed that any record type other than 1 or 2 will be treated in the same manner as record type 5.

Example: The object program:

```
                                 ORG     7000H
        7000  210000     START   LD      HL,0
        7003  AF                 XOR     A
        7004  C9                 RET
        7000                     END     START
```

would be stored in a single disk sector as

```
byte
offset              Hexadecimal Data

00      0107 0070 2100 00AF C902 0200 7000 0000
10      0000 ...
.       ...
```

F. FORMAT OF SYSTEM TAPES

A "SYSTEM" tape is created by the ASMBLR program when the OBJECT TO TAPE option is selected. A "SYSTEM" tape contains object code in a format which is loadable under the SYSTEM command of level 2 and disk BASIC.

All TRS-80 tape files begin with a synchronization leader which is comprised of 128 zeros. The "SYSTEM" file is no exception. The synchronization leader is followed immediately by a byte containing 55H. This is followed by 6 bytes containing the tape filename in ASCII. If the filename is less than 6 bytes long then it is left justified and padded on the right with blanks.

The filename is followed by one or more variable length records which specify the loading of object code. Each of these records begins with a byte containing 3CH. The next byte specifies the number of bytes of object code to be loaded. The next two bytes contain the address at which to load, in standard lsb-msb format. These are followed by the object code bytes themselves. Finally, each record contains a checksum byte. This byte is calculated as the arithmetic sum (with carry ignored) of each object code byte in the record, as well as the two load address bytes.

The last object code record is followed by a byte containing 78H. This indicates the end of the tape file. It is followed immediately by the transfer address in standard lsb-msb format.

Example: The object program:

```
                            ORG      7000H
     7000  210000    START  LD       HL,0
     7003  AF               XOR      A
     7004  C9               RET
     7000                   END      START
```

would appear on a "SYSTEM" tape named TEST3 as the following sequence of bytes (shown in hexadecimal):

```
<sync> 55 54 45 53 54 33 20 3C 05 00 70 21 00 00 AF C9 09 78 00 70
```

G. HEXCONV SUBROUTINE PACKAGE

The HEXCONV subroutine package has been included in the M-ZAL system for demonstration purposes and can also be of general use to the assembler language programmer. The package consists of 4 hexadecimal conversion subroutines which are all contained in module HEXCONV. The routines are:

HA2CONV - convert contents of reg A to ASCII characters representing the hexadecimal equivalent. Place result in two byte buffer area pointed to on entry by reg HL. Increment HL by two and destroy AF.

HA4CONV - convert contents of reg DE to ASCII characters representing the hexadecimal equivalent. Place result in four byte buffer area pointed to on entry by reg HL. Increment HL by four and destroy AF.

AH2CONV - parse two ASCII characters representing hexadecimal digits into a binary value in reg A. Input string is pointed to upon entry by reg HL. HL is incremented by two. Flag status "C" is returned if the input string is invalid, otherwise flag status "NC" is returned.

AH4CONV - parse four ASCII characters representing hexadecimal digits into a binary word in reg DE. Input string is pointed to upon entry by reg HL. HL is incremented by four. Flag status "C" is returned if the input string is invalid, otherwise flag status "NC" is returned. Register AF is destroyed.

Example:

```
                EXTRN   HA4CONV
        START   LD      DE,(VALUE)      ;VALUE TO DISPLAY
                LD      HL,3C00H        ;SCREEN LOC TO DISPLAY AT
                CALL    HA4CONV         ;DISPLAY "VALUE" ON
        ;                                SCREEN IN HEXADECIMAL.
                RET
        VALUE   DEFW    28
```

This code will cause the character string "001C", corresponding to the hexadecimal equivalent of 28, to be displayed in the top left hand corner of the video display.

H. DECCONV SUBROUTINE PACKAGE

The DECCONV subroutine package has been included in the M-ZAL system for demonstration purposes and can also be of general use to the assembler language programmer. The package consists of 2 decimal conversion subroutines, both of which are contained in module DECCONV. The routines are:

DACONV	- convert the contents of HL into a five digit unsigned decimal number, represented in ASCII and placed in the buffer area pointed to upon entry by reg IX. Increment reg IX by five and alter no other regs.
ADCONV	- parse a variable length ASCII character string representing an unsigned decimal value into a binary word in reg DE. The input string is pointed to upon entry by reg HL. It can be from one to five characters long and must be terminated by a non-numeric character. Reg HL will be incremented to point to the terminating character. Reg AF is destroyed. Flag status "C" is returned if the input string is invalid, in which case reg DE is not altered. Otherwise flag status "NC" is returned and the value is returned in reg DE.

Example:

```
        EXTRN   DACONV
START   LD      HL,(VALUE)      ;VALUE TO DISPLAY
        LD      IX,3C00H        ;SCREEN LOC TO DISPLAY AT
        CALL    DACONV          ;DISPLAY "VALUE" ON
;                                SCREEN IN DECIMAL.
        RET
VALUE   DEFW    8000H
```

This code will cause the character string "32768", corresponding to the decimal equivalent of 8000H, to be displayed in the top left hand corner of the video display.