DOW EDITOR/ASSEMBLER

For the Texas Instruments 99/4A Home Computer with the Mini Memory Command Module

Requires TI99/4 or TI99/4A with cassette cable and recorder and the Mini Memory Command Module.

Highly recommended: the Editor/Assembler Manual from Texas Instruments.

Disk drive and printer may be used but are not required.

TABLE OF CONTENTS

INTRODUCTION

This manual was written with the assumption that you, the reader, are reasonably familiar with assembly language on the Home Computer. That is, this manual is not a tutorial, nor is it a reference manual for the Home Computer.

You should become thoroughly familiar with the Mini Memory Command Module's Manual and with the Editor/Assembler Manual, both from Texas Instruments, for a technical description of the computer and its instruction set.

To help you understand how to use the *DOW* Editor/Assembler, read first Section 2 of this manual to put the assembler in proper perspective, as compared to the two assemblers from Texas Instruments. Then read Sections 3 through 6 for an explanation of syntax, commands, and directives. Sections 7 through 9 give instructions for actually writing a program while section 10 presents a complete example program. Section 11 discusses how to handle large programs. Section 12 lists and explains the error messages. Finally, section 13 describes some minor changes you may wish to make to customize the DOW Editor/Assembler to your own Home Computer's peripherals.

COMPARISON OF ASSEMBLERS

The DOW Editor/Assembler complements the Line-by-Line Assembler and the Editor/Assembler products from Texas Instruments. Whichever assembler you use, you will need TI's Editor/Assembler manual as a technical description of the Home Computer. The differences between the three assemblers are, briefly:

1) The Line-by-Line Assembler uses the Mini Memory Command Module. It runs in the Module's 4K RAM, using roughly 75% of it. It will remain in the Mini Memory Command Module with your assembly language program, if your program is small enough. In addition to the manual and module, you only need a cassette tape recorder and the cassette cable (so you can load the assembler into the module's memory.)

2) The DOW Editor/Assembler also uses the Mini Memory Command Module. Like the Line-by-Line Assembler, you only need a cassette tape recorder and cable, the manual, and the module to use it. However, it is in fact a Basic program, which means that it executes out of the 16K RAM in the console itself. This leaves all 4K RAM in the module available for your assembly language program. This also means that it has more features. For instance, you can edit your unassembled program, including operations such as changing statements, deleting statements, inserting new statements (even in the middle of the program), saving the program on cassette, and listing the program if you have a printer. In short, you can develop a program much the same way you do in Basic, without having to type it in

again or patch it in hex.

The DOW Editor/Assembler will assemble all instructions for the 99/4 or 99/4A processor. It also has several assembler directives (DATA, BYTE, TEXT, BTXT, BSS, EQU). It does not recognize RT (use B *R11 instead) or NOP.

3) The Editor/Assembler does not use the Mini Memory Command Module but has its own module. To use it you need the Peripheral Expansion System with memory and disk storage; this of course makes it more powerful than either of the other two assemblers, but also it requires substantially more investment in computer equipment to use it.

SYNTAX

Statements are entered in standard format, with just a few exceptions noted here.

Labels are one to three characters: the first must be one of the letters A-Z, the others can be A-Z or 0-9. There can be no more than 40 labels. A colon separates the label from the statement.

The forms $+n and $-n are not allowed, although these are typical forms for jump instructions with other assemblers. An example of what is not allowed might be: JMP $+20, meaning to jump down 20 bytes in the program. With the DOW Editor/Assembler, all jumps must refer to a labeled statement.

Do not use commas; use semicolons instead.

Do not use quotation marks.

Remarks on any statement are allowed, following at least one blank space.

The assembler works with even byte addresses, so there is an implied EVEN statement following each BYTE, TEXT, or BTXT statement.

The longest statement allowed is 76 characters.

COMMANDS

The prompt for a command is "---->". The commands are listed and described below.

---->STOP

Stops the DOW Editor/Assembler.

---->NEW

Initializes for a new program. (This need not be done when the program is first run.)

---->OLD

Restores assembly language program from cassette. (It is not necessary to use the NEW command first, even if you already had a program in memory.) Before the program is read in, you will be shown the title for the program on cassette so you can verify that it is the correct program. See Section 13 if you wish to use with a disk drive.

---->SAVE

Saves assembly language program on cassette. The program remains in memory. The TITLE command is automatically called; see below. See Section 13 if you wish to use with a disk drive.

---->LIST

Lists the program and its title at the thermal printer. The TITLE command is automatically called; see below. See Section 13 if you have a different printer.

---->TITLE

Shows old title, if any, and gives you the option to change it. A title is whatever you want it to be to help to identify the program listings. You may wish to put a revision date and the name of the cassette on which the program is stored in the title. The maximum number of characters in a title is 180.

You will be shown the previous title, if any, with the label "OLD". You are then asked if it is "OK"; if you do not wish to change it, enter "Y". Otherwise the prompt "NEW" will appear to ask you to enter a new title.

Each LIST or SAVE does the TITLE command for you automatically.

---->LOAD hhhh

Loads the program at hex address hhhh. Example: LOAD 7600. You will be asked to confirm the loading address before loading starts. After the program is loaded, you are shown the address of the next available memory location. Note: EQU's at the end of the program are not included when computing the next available address. Any label errors are found by this command. The LOAD process can take ten minutes for a large program, so to let you monitor the progress a "." is displayed for each word (that is, for every two bytes) loaded. Before the first "." is displayed, there is a pause while the program is scanned to assign locations to any labels.

There is no AORG directive, as there is with TI's assemblers. The program is loaded starting at the location specified by the LOAD command. Make sure you specify an even address.

---->LINK name

Calls your assembly language program by means of the Basic subroutine LINK, as though your program were a subroutine having no arguments. If you wish to test a program segment before including it in a larger program, you can pass data to and from it through specific memory locations by using the MINI command (see below). You must prepare the User Defined *REF/DEF Table, which starts at the location* pointed to by LFAM (701E) and ends at >7FFF, before you call your program. (See Section 7.)

If the name specified on the LINK command is not found in the table, the DOW Assembler/Editor will abort. In that case, you will have to use the Basic RUN command again. Also, the source version of your assembly language program will vanish from VDP RAM, so prudence dictates that you SAVE and LIST (if you have a printer) your program after making major changes, and that you type the LINK command very carefully. To return from your program, use the B *R11 instruction. Make sure your program does not alter VDP *memory other than a Basic program would alter* it; this precaution must be observed so that you can safely return to the DOW Editor/Assembler.

---->MINI hhhh

Displays or alters memory locations. Specify the location in hex; eg, MINI 7FF8. (Although the EASY BUG program development tool available with the Mini Memory Command Module also allows you to display or alter memory, you have to wipe out the DOW Editor/Assembler and your program in VDP RAM in order to use it.) The display shows the location address in

hex, the value at that and the next location (also in hex), and prompts for your action with "?". You have three choices:

1) enter a period to exit from the MINI command.

2) just press ENTER to move down two locations.

3) enter a new value in hex or positive or negative decimal to replace the value in memory. In this case, the same two locations are shown to you again so you can verify that the correct change was made.

You can use the MINI command to convert a decimal value to hex. Just put the value into some location in memory which you know can be changed without harm.

Example: to convert 4029 to hex, do this...

```
---->MINI 7200          (Assuming 7200 is free)
>7200 >hhhh ?4029       (hhhh is current value)
>7200 >0FBD ?.          (FBD is hex equivalent)
```

---->EDIT

Enters edit mode, which allows you to enter or change your program's source statements. In edit mode, the prompt is "E->". There are several commands in edit mode. They are described in the next section of this manual.

EDIT MODE COMMANDS

After typing the EDIT command, you can use one of the following commands.

1) "." Typing a period causes you to leave EDIT mode.

2) Positioning and verifying commands. (The commands are underlined below.)

2a) "T" Typing a "T" repositions at location 000.

2b) "Thhh" Typing a "T" followed by a hexadecimal number repositions at that location. Example: T5E repositions at 05E.

2c) "U" Typing a "U" repositions up one line.

2d) "Un" Typing a "U" followed by a decimal number repositions up that many statements.

2e) "D" Typing a "D" repositions down one line.

2f) "Dn" Typing a "D" followed by a decimal number repositions down that many statements.

2g) "V" Typing a "V" verifies the current line by displaying it at your monitor. (The current line is always verified after all commands except T, U, D, and V.)

2h) "Vn" Typing a "V" followed by a decimal number verifies that many statements. The last statement shown is the current line.

As a matter of convenience, two or more of these commands may be joined together. Any blanks between them are usually ignored. For instance, you could type "T2E" and "V6" as "T2E V6". To determine the last instruction in the program, type "TFFF V". As another example, having just made several changes to your program, you might want to review that part with "U9 V10".

Except for the "T" command, the space between commands is not necessary. If you wish to follow the "T" command with the "C", "D", or "E" comand, you must put a "V" or a space between them because the letters C, D, and E would otherwise not be treated as commands but as part of the hexadecimal number following the "T" command. Example: use "T18 C", not "T18C".

3) "X" Typing an "X" deletes the current statement. In so doing, any following statements are automatically shifted up in memory.

4) "?" Typing a question mark computes the space remaining. If your program is large, do this periodically. If the number of bytes remaining free goes negative, you will probably lose your program when trying to SAVE it or to LOAD it.

5) "C" Typing a "C" allows you to change the current instruction. You will first be asked for the part to be changed, then you will be asked to indicate what should replace it. In this example, the "7" of "76" is changed to "10" to make "106".

```
08A A   :LI    R9;76       R9=# OF SHIPS
E->C                  (Change command.)
FR:7                  ("From" string.)
TO:10                 ("To" string.)
08A A   :LI    R9;106      R9=# OF SHIPS
```

If the statement has an error after the change is made, the statement is restored to its original form. If the change is made and the number of locations used by the instruction changes, any following instructions are automatically moved up or down in memory. When entering a "from" or "to" string that begins or ends with one or more blanks, you have to enclose the string in quotation marks because Basic trims blanks off either end of a string. For example, do this to put the label TOP on a statement which does not have a label:

```
FR:"    "                   (four blanks)
TO:TOP:                     (new label)
```

If you enter a null string for FR:, the TO: string will be put in front of the instruction. If you enter a null string for TO:, the FR: string will be deleted.

6) "E" Typing an E puts the editor into enter mode. This mode is used to enter (or insert) new statements into your program. If no statements have been entered already, it will start at location 000. If some statements have already been entered, the new statement(s) will go immediately after the last one displayed on the monitor. Thus, you may enter either in the middle of the program or at the end (but not before the first statement). As you enter statements, any that follow will be automatically shifted down in memory.

For example:

```
E->E                              ("Enter" mode)
LOC LBL:OPCD OPERAND(S)  (Column alignment aid.)
05E x                             (Your prompt)
```

(The x marks the position of the cursor.)

Your prompt is the three digit hexadecimal number which is the location the instruction will have. Each statement is always loaded at an even address, so the number displayed is always even.

The column alignment aid is a line printed to show you in which columns you should enter the various fields within the statement. The "LBL:" shows the columns in which you put a label and the colon which must follow it. Enter four spaces if you do not want a label. Under "OPCD" you enter the operation code, then space over to "OPERAND(S)" to type any operands. You may enter a remark after the operand(s), providing that you leave at least one space before it. If the instruction has no operand, start the remark under or to the right of the "P" of "OPERAND(S)". Note that you must not use a comma or quotation marks,

since your input is read as a string by the Basic interpreter. If the instruction has two operands, use a semicolon for the separator.

To exit from enter mode, simply type a null line. You will return to the normal edit mode and be prompted with "E->".

7) "R" Typing an "R" replaces the current statement with one you type in. If the new statement has an error and does not assemble, the original statement is not replaced. If it is replaced and the number of locations used by the new instruction is different, any following instructions are automatically moved up or down in memory.

ASSEMBLER DIRECTIVES

These directiyes are entered and edited the same way that instructions are. Labels and remarks may be used.

DATA

This directive loads data into memory. Values can be hexadecimal, positive decimal, or negative decimal. More than one value can be specified. Example:

```
    DATA >2000   MASK
X   :DATA >F9A2;-20;1000
```

BYTE

This is similar to DATA except that the values are single byte values. Example:

```
    BYTE >FD;17
```

Negative decimal values are not permitted; use the single byte hex equivalent instead. Note: even though an odd number of bytes may be specified, the next instruction will be at an even address (relative to the address specified when the program or program segment is LOADed).

TEXT

A string of characters is loaded. The string is set off by a beginning and ending break character. Use any character you want. Example:

```
    TEXT /HELLO/   THIS IS A REMARK
    TEXT .GOOD-BYE.
```

Just as with BYTE, the next statement will start at an even location even if an odd number of bytes is specified.

BTXT

If your program is to be called from Basic (eg is called via the LINK command), any text characters have to be biased by >60. The BTXT directive is identical to the TEXT command except that each character is biased by >60. For instance, a blank will generate >80 instead of >20.

BSS

You can reserve up to 512 bytes of space in your program with BSS. For example. to reserve 2 words, or 4 bytes:

```
    BSS  4
```

The values already in these locations are not changed when the program is loaded.

Suggestion: for large areas of memory, simply decide on an area of memory and use EQU directives to refer to it; make sure that you do not let such data areas overlap with your program or other data areas.

EQU

This directive allows you to use a label to refer to an address. For instance, to load R1 with the address of a particular region of memory (say 7C00):

```
    LI   R1;BUF
    (other instructions here)
BUF:EQU  >7C00   TEMP SCREEN IMAGE TABLE
```

Section 6: ASSEMBLER DIRECTIVES

As another example, to call VDP Multi Byte Write (VMBW):

```
     BLWP @MBW
     (other instructions here)
MBW: EQU  >6028   VMBW
```

Some advantages of using EQU's are:

1) There can be several references in a program to the same value but it need only be defined once. This makes it easy to change.

2) It is easier to key in the label than the 4 digit value.

3) If you group all the EQU's at the end together, it is easy when reading the program to see how it interacts with the rest of the system. For instance, you can see which utilities it uses, which regions of memory you have defined, or which of your own routines it uses.

4) It is easier to put a remark on one EQU statement to identify its value than it would be to identify the value each time it is used.

Note that an EQU appears to occupy two bytes of memory in your program. However, the LOAD command does not load anything into the Mini Memory Command Module for those two bytes. Furthermore, if the EQU comes at the end of the program, the LOAD process will not include the bytes in its computation of the next available memory location. For these reasons, the best place for EQU's is at the end of your program.

Section 6: ASSEMBLER DIRECTIVES

THE REF/DEF TABLE

To be able to call a program, its name must be loaded into the User Defined REF/DEF Table. See the Mini Memory Command Module Manual for a description of the table.

In the two examples below, the name of the program is "DEMO ", it is the only program in the table, and the program is loaded at 7500.

Before keying in the program, if you know what its entry point will be you can create its entry in the User Defined REF/DEF Table as follows:

```
---->MINI 701E          (Set LFAM to 7FF8)
>701E >hhhh ?>7FF8      (hhhh is current value)
>701E >7FF8 ?.
---->EDIT
E->E
LOC LBL:OPCD OPERAND(S)
000      TEXT /DEMO   / (Name is DEMO)
006      DATA >7500     (EP at 7500)
008                     (Just press ENTER)
006      DATA >7500     (Shows last entry)
E->.                    (Exit from edit mode.)
---->LOAD 7FF8          (Load into table now.)
ADDR = >7FF8  OK?Y
....
NEXT = >8000
---->NEW                (Get ready to enter)
---->EDIT               (Now enter the program)
E->E
```

The above technique cannot be used if you have already keyed in your program (unless you SAVE it first). The method below may be used if you do not wish to destroy your program.

The drawback with this second method is that it requires you to know the name of your program in hex.

```
---->MINI 701E          (Set LFAM to 7FF8)
>701E >hhhh ?>7FF8      (hhhh is current value)
>701E >7FF8 ?.
---->MINI 7FF8
>7FF8 >hhhh ?>4445      (This loads "DE")
>7FF8 >4445 ?           (Just press ENTER)
>7FFA >hhhh ?>4D4F      (This loads "MO")
>7FFA >4D4F ?           (Just press ENTER)
>7FFC >hhhh ?>2020      (This loads 2 blanks)
>7FFC >2020 ?           (Just press ENTER)
>7FFE >hhhh ?>7500      (7500 is program's
>7FFE >7500 ?.           entry point)
```

RECOMMENDED SEQUENCE

Below is a sequence of instructions you might want to follow when working on a program. First, follow the examples in Section 7 to make an entry in the REF/DEF table. Then proceed as below:

A) OLD — You would do this only if your program had been saved earlier.

B) EDIT — To enter or change your program.

C) ? — If your program is large, use this edit command often to see how much memory is still free. If not enough, do not continue since you may lose your program. Cut down on remarks to gain more free space.

D) LOAD — There are two reasons for loading before SAVE, LIST, or LINK.
1) It might find a label error.
2) Do it right away so you do not forget to do it later.

E) SAVE — In case something goes wrong with LINK. Be especially careful to do this after making many changes.

F) LIST — This is always a good idea, if you have a printer. As with the SAVE, do this especially after making many changes.

G) LINK — Now give it a try.

H) Go back to step B to make any changes.

CALLING FROM BASIC

Several things must be remembered for your program to return successfully to a Basic program. This includes the DOW Editor/Assembler when your program is called with the LINK command.

Make sure interrupts are turned off. That is, if you enabled them with the LIMI 2 instruction, disenable them with LIMI 0.

Return with a B *R11 instruction. However, if you wish to use register 11 in your program (perhaps to call subroutines within your program), you should first move the return address to some other register (or location) and return via that register (or location).

Make sure you clear location >837C, the STATUS byte, before returning. This is set by various calls to GPLLNK.

PROGRAM EXAMPLE

This program demonstrates the use of Assembly Language and the DOW Editor/Assembler to talk to a user at the console. As such it is representative of part of a larger program, which presumably would perform computations or elaborate graphics or sound effects. Although what it does is not flashy, it shows you some important techniques you will need to know.

Remember that you will have to refer to both Texas Instruments manuals to be able to understand this program. These manuals are the one supplied with the Mini Memory Command Module and the Editor/Assembler manual, sold separately.

After clearing the screen, the program prompts the user for an input number with "DATA:" and by beeping. If characters other than digits are entered, the bad response tone is given. Once the ENTER key is pressed, the value is stored in location >7200 and the program returns to the Basic program which called it. (You can use MINI 7200 to check what it did.) Before entering the program, read Section 7 for two easy methods of preparing the REF/DEF table.

The program is listed in parts below. Following each part is a paragraph or more of explanation.

```
DEMO PROGRAM - INPUT AND DISPLAY
INTEGER VALUE (LOAD AT 7500)
000     LI   R1;766  CLEAR SCREEN
004     LI   R2;>8080 (BLANKS)
008 TOP:MOV  R2;@BUF(R1)
00C     DECT R1
00E     JOC  TOP
010     CLR  R0      WRITE
```

```
012     LI   R1;BUF  BLANKS
016     LI   R2;768  TO
01A     BLWP @MBW    SCREEN.
```

The first two lines printed are the program title. The program should be loaded at >7500, as the title indicates.

The first nine instructions blank out the screen by filling VDP locations 0 through 767 with blanks (>80). Note: >80, not >20, is a blank because when Basic is running all display characters must be biased by >60. The 768 blanks are first loaded into CPU RAM, starting at location >7200. They are then written into VDP RAM.

The JOC operation is useful in conjunction with DEC or DECT if a loop should continue until less than 0. To stop at 0, use JGT.

The call "BLWP @W" (at 010 through 01A) uses VMBW to write the 768 blanks to the screen to clear it.

```
01E     LI   R0;392  WRITE PROMPT
022     LI   R1;PRO  TO
026     LI   R2;5    SCREEN.
02A     BLWP @MBW
02E     LI   R0;397  INPUT POS.
032     CLR  R2      R2=NUMBER.
034     MOVB R2;@MOD MODE 0.
038     BLWP @GPL    ACCEPT
03C     DATA >34     TONE.
```

Locations 01E to 02A write "DATA:" to the screen at position 392. At 02E, R0 is set to point to the input position, which is immediately to the right of the prompt message on the screen. Register 2 is cleared to accumulate the number to be entered. The 0 now in R2 is used to set the mode to 0 for the

call to KSCAN which follows below. At 038-03C, the accept tone is started.

```
03E LP :LIMI 2          ALLOW INTER-
042     LIMI 0          RUPTS BRIEFLY.
046     BLWP @KEY       CALL KSCAN.
04A     MOVB @STA;R1    CHECK STATUS
04E     COC  @MSK;R1    FOR NEW KEY.
052     JNE  LP         NOT YET.
```

The input loop starts at label LP. First, interrupts are allowed briefly. This is necessary so a tone can be generated. Then KSCAN is called. The status byte is moved to R1 so bit 2 can be tested. If it is set, a key has been pressed. If not set, loop back to LP.

```
054     CLR  R1         YES.
056     MOVB R1;@STA    CLR STATUS.
05A     MOVB @INP;R1    LOOK AT IT.
05E     SWPB R1
060     CI   R1;>D      ENTER?
064     JEQ  END        GO IF DONE.
```

R1 is cleared so the status byte can be reset to 0. Then the value for the key which was pressed is moved into R1 and swapped to the right, or low order, byte. At 060, a check is made to see if the ENTER key (>D) was pressed; go to END if so.

```
066     AI   R1;-48     NO. CHECK
06A     JLT  ERR        FOR DIGIT.
06C     CI   R1;9
070     JGT  ERR
072     MPY  @V10;R2    OK. COMPUTE
076     MOV  R3;R2      NUMBER
078     A    R1;R2      IN R2.
```

If not ENTER, continue by subtracting >30 to turn the key code into its value, if it is a

digit. (Subtract >30 by adding immediate -48.) If the result is less than 0 or greater than 9, go to ERR because it was not a digit. If a digit, multiply the number so far (in R2) by 10. This puts a product in registers 2 and 3. Move the low order half of it, in R3, back to R2. Then add the latest digit, in R1, to the total, in R2.

```
07A       AI   R1;>90   NOW WRITE
07E       SWPB R1       DIGIT TO
080       BLWP @SBW     SCREEN.
```

Bias the digit by >60 so it can be displayed and by >30 = 48 (because it was subtracted out above). In other words, add >90 to it. Then swap it to the left byte and use VSBW to write it to the screen.

```
084       INC  R0
086       JMP  LP       GO FOR NEXT.
```

Increment the screen position pointer and loop for another digit.

```
088 END:  MOV  R2;@>7200 STORE R2.
08C       B    *R11     BACK TO BASIC.
```

Terminate by returning to Basic. The accumulated value is first stored at >7200. It is safe to return because we cleared the status byte after the last KSCAN. (If the byte is not cleared, a false error can be reported.)

```
08E ERR:  BLWP @GPL     ERROR.
092       DATA >36      BAD TONE.
094       JMP  LP
```

At ERR, the bad response tone is generated.

Section 10: PROGRAM EXAMPLE

```
096 PRO:BTXT /DATA:/
09C MSK:DATA >2000    MASK.
09E V10:DATA 10       VALUE TEN.
0A0 MOD:EQU  >8374    MODE.
0A2 BUF:EQU  >7200    BUFFER.
0A4 KEY:EQU  >6020    KSCAN.
0A6 STA:EQU  >837C    STATUS.
0A8 INP:EQU  >8375    KEY PRESSED.
0AA GPL:EQU  >6018    GPLLNK.
0AC SBW:EQU  >6024    VSBW.
0AE MBW:EQU  >6028    VMBW.
```

Finally, data and equates end the program. The BTXT directive is used because the program is called from Basic. The EQU's come at the end of the program; the next available address reported to the programmer by the LOAD command will be 75A0.

After entering the program (and correcting any errors), use LOAD 7500 and LINK DEMO to run it. (But first, SAVE and LIST.)

PROGRAM MODULARITY

A restriction of the DOW Editor/Assembler is that your program must fit into 512 byte segments. If it is small enough to fit entirely into 512 bytes, there is no problem. That is of course 256 words, or due to the fact that many instructions are two or more words in length, about 150 instructions.

When you are editing your program, statements are identified in the left margin by their relative location within the program segment. This value can range from 000 to 1FE. This is the 512 byte limit.

If you break a large program into segments, you may find it convenient to load the segments at nice, even addresses, such as 7200, 7300, 7400, and so forth. This will make it easier to change and reload a segment, because you simply have to make sure that the last address to be loaded in each segment does not exceed 0FE. If you wish, you may allow each segment to grow to the full size of 1FE; in that case, load the segments at 7200, 7400, 7600, and so on. By using these addresses it is easy to inspect or patch the program since the computation of the actual address (known as the absolute address) consists of adding a value such as 7200 to a two or three digit relative hex address such as >1B6.

Each time you lbad a segment into the Mini Memory Command Module's 4K RAM, it will stay there indefinitely. This means that you can load the segments as you write and test them, even if this process is spread out over hours, days, or weeks.

In order for this type of program writing to work well, you have to design your large

program so that it consists of meaningful modules. (Here the term "module" refers not to "Command Modules" but to software modules, or program segments.) It is important that each module make sense by itself. That is, do not simply divide a large program into chunks of approximately 512 bytes each.

The various modules of your program are like building blocks. Each should be strong. Strength means that everything in the module belongs together for logical reasons.

It is also important that the links between the modules are not strong. Modules are weakened when things which should be together in a module are distributed throughout more than one. They are also weakened by having to share too much data. Ideally, when one module calls another, as little information as possible should be passed because this makes each module easier to understand, to write, and to test by itself.

Data may be passed (or shared) between modules by using EQU's to define the same areas in memory, but try to keep the number of locations to a minimum and use each location consistently in all modules. Try to pass all information through registers if possible.

Above all, do not have one module "know" how another works. Each should be treated as a "black box" which performs according to external specifications. You should be able to write and test your modules independently.

A large program that is written by this type of modular approach is much more apt to work than if it is developed as one very large and very complicated program.

Section 11: PROGRAM MODULARITY

When a large program is broken into modules, you have to specify how to get from one to the other. A good overall scheme is to think of one of the modules as a main program. The main program contains the overall logic and makes it happen by calling upon subprogram modules, using the BL instruction. Thus, by reading just the main program you can check the sequence of major events in your program. Use the EQU directive to enable the main program to refer to the subprograms. The subprograms should be loaded at nice locations (such as 7400) with the very first location of each being the entry point. In this way the EQU's in the main program will have values such as 7400, 7600, etc.

In addition to a main program and subprograms (each of which may only be called once from the main program), you might want to write a number of functions or subroutines. Usually these are rather small and well defined routines that are used a number of times. Examples would be generating a sequence of tones or clearing the screen. You could put a number of these together into one module - a subroutine library. This reduces the number of cassette tapes needed to save them. It also makes it easy to pack them into a block of memory. To do this neatly, you should build a "transfer vector" into the front of the library module.

The example below shows the outline for a module having several library subroutines (SIN, COS, and TAN):

```
     JMP   SIN      (This is the
     JMP   COS       transfer
     JMP   TAN       vector.)
SIN:...             (Do SIN here)
     BL    *R11     (Return)
COS:...             (Do COS here)
     BL    *R11     (Return)
TAN:...             (Do TAN here)
     BL    *R11     (Return)
```

Let us assume that this module has been loaded at 7C00. A program needing to use SIN, COS, or TAN would equate them as follows:

```
SIN:EQU 7C00
COS:EQU 7C02
TAN:EQU 7C04
```

Because the Mini Memory Command Module will hold your subroutine library indefinitely, you can build it and test it, then use it again and again within one program. You can continue to use it even if you write different programs at different times, as long as you do not reuse the same memory space.

ERROR MESSAGES

A "bad response" tone is given for all errors.

1) Commands and instructions. You will see **"ERROR"** displayed. An up arrow "^" is displayed under the point where the error was discovered. This is usually at or immediately to the right of the offending character.

2) A **BSS statement** that implies an address past the 512 byte program limit is indicated with a "^" after the byte count you entered.

3) If the **break character** is not found for TEXT or BTXT, the "^" is displayed under the first break character.

4) **OVERFLOW** You get this when editing and you try to enter beyond location 1FF. You also get it by entering or changing an instruction which would cause the last instruction to go past 1FF.

5) When using the "C" command in edit mode, if the "from" string is not found, you will be told **"NOT FOUND"**.

6) When loading, either an undefined label or a multiply defined label is reported by:

"_ERR: LBL xxx AT hhh"

where xxx is the label and hhh is the relative location where the error was discovered. The load process stops when the error is discovered.

7) A statement that is longer than 76 characters on entry (or after being changed) is reported as **"TOO LONG"**.

CUSTOMIZING

You can use a printer other than the Texas Instruments Thermal Printer (TP). For example, if you have the Texas Instruments Impact Printer, change the TP in statement 1650 to RS232.BA=4800 (assuming that you have the baud rate set at the recommended 4800).

If you have a disk drive and wish to load the DOW Editor/Assembler from disk, be sure always to type in the following statements before loading it:

```
CALL FILES(1)
NEW
```

Also, to use with a disk drive, you must change the value 3300 in statement 1510 to be 2250; this reflects the loss of memory due to having the disk drive on. You will see this loss when you use the "?" command in edit mode.

If you wish to save and load your assembly language programs on disk, insert these two statements:

```
2219 INPUT "FILE=":FILE$
2369 INPUT "FILE=":FILE$
```

Then change "CS1" (including the quotation marks) to FILE$ in statements 2220 and 2370.

To list to a disk file (so you can read it with the editor in TI's Editor/Assembler), change "TP" in statement 1650 to FILE$ and insert:

```
1649 INPUT "FILE=":FILE$
```