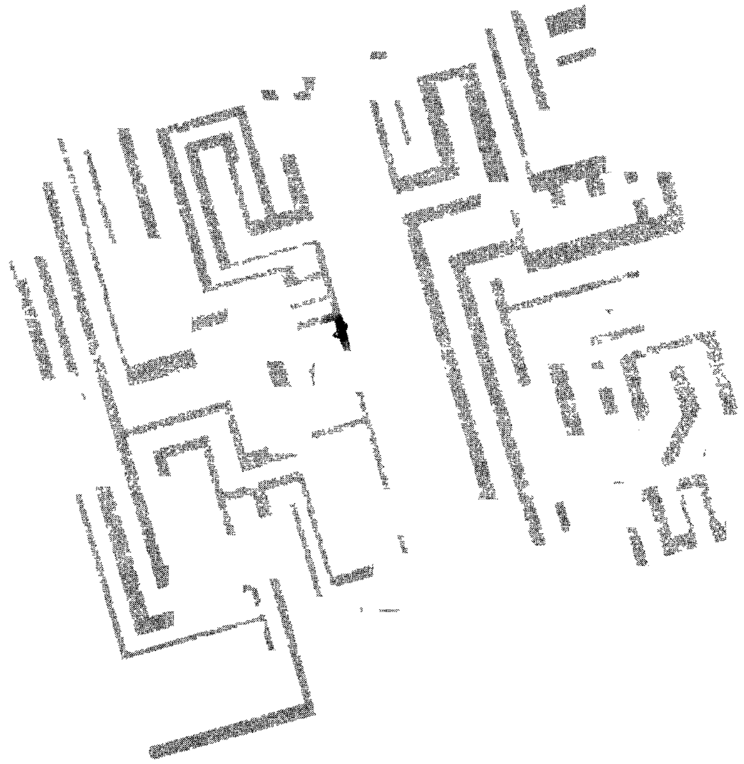
Introduction to Assembly Language for the TI Home Computer

by Ralph Molesworth



Edited by Steve Davis

Introduction to Assembly Language for the TI Home Computer

By Ralph Molesworth

FIRST EDITION

Copyright © 1983 by Steve Davis Publishing

Steve Davis Publishing
P.O. Box 190831
Dallas, Texas 75219

All Rights Reserved

ISBN 0-911061-01-0
Library of Congress Catalog Card Number 83-90770

TABLE OF CONTENTS

CHAPTER ONE

The purpose of this book, as the title implies, is to introduce you, the TI 99/4 and 99/4A Home Computer user, to TMS9900 assembly language and get you started writing in assembly with a minimum of grief. The reference guides provided with the TI Editor/Assembler and Mini-Memory packages are just that, reference manuals. They assume you have a knowledge of programming in assembly. Naturally, this poses a problem for the beginner. Now, this book can act as a supplement to give you the background and skills needed to understand and utilize your Editor/Assembler manual. Upon completion of this material, you will be able to code simple programs in 9900 assembly language and be better prepared to use your Editor/Assembler, or the Line-by-Line assembler with the Mini-Memory, and the reference manuals to develop more complex programs and routines.

It is best to read this book from start to finish because each lesson builds upon the previous one and program examples illustrate functions covered to that point. To get the most from this book, you should have the TI Editor/Assembler software, which includes the Editor/Assembler manual. Those who do not own the peripherals necessary to run that software (memory expansion and disk drive) may use the Mini-Memory module to enter many of the program examples. In this book, when the Line-by-Line assembler is mentioned, this will refer to the assembler program that is provided with Mini-Memory. The Line-by-Line assembler has certain restrictions which are described later in this book, but the concepts of writing assembly programs is the same as with Editor/Assembler. Those without Editor/Assembler should obtain a copy of the TI Editor/Assembler manual, which is available separately. At the end of each section, this book provides page references to that manual where relevant information can be found.

Before proceeding with this work, you should be familiar with using TI BASIC. Just as the BASIC language varies slightly from one type of computer to another, so do assembly languages. But if you are able to learn one form of the language, it is simple to apply what you have learned to another similar language. By now, you should have experience writing at least simple programs in TI BASIC and TI Extended BASIC. You will see that assembly is far more powerful than BASIC. But, it is a computer language, and like all computer languages it has particular rules, disciplines and terms that must be learned. Once you learn more than a couple of computer languages, the similarities between various languages will be more apparent than the differences.

TAKING THE PLUNGE FROM BASIC TO ASSEMBLY

The program statements you probably have written so far in TI BASIC have no meaning to the computer taken as they are. Each statement must be interpreted into the language the computer understands. This is called "machine language." A language which is at machine level is said to be a "low level" language. A language such as TI BASIC which closely resembles English phrases is called a "high level" language. The interpretation process is carried out by a system of programs, subroutines, and data put into your computer by the manufacturer. This system is known as the BASIC interpreter.

The interpreter analyzes each BASIC statement and converts the high level BASIC statements into a set of machine language instructions which actually cause the computer to function. No matter what popular programming language you write in, they all must be translated into the machine code the computer understands. It is the need to perform this interpretation process as each statement is executed that causes programs written in TI BASIC or TI Extended BASIC to run much slower than a program comprised of machine code instructions.

The assembly process improves upon this vastly. A program written in assembly language is assembled by the assembler much the same as the interpreter processes a BASIC language program. However with assembly, the resulting set of machine codes can be saved on some device for reuse. Then, when the program is run, there will not be any "middleman" as in BASIC. The machine code is directly executable by the computer and, so, executes very fast. Many arcade type games and some application programs written in TMS9900 assembly language actually require subroutines to waste time in order to slow them down!

The statements which comprise a program written in a high level language such as TI BASIC are merely the source of material from which a machine language program is generated. In assembly language, the program that you code is called the source program. The program which is assembled from your source code is called the object program. Since these are always saved on some device, they can also be called the source file and the object file. After your assembly language program has been assembled, it exists in two forms. The original source code remains unchanged while a new file containing the object code is created. (The Line-by-Line assembler, as the name implies, creates machine code when you enter the line. Thus, your source code is not saved as a separate file).

In addition to the redundant interpretation of a BASIC language program during execution, there are other drawbacks as well. The BASIC language program needs a large and sophisticated "supporting cast" of hardware and software just to make it all work. This requires that a portion of the computer's resources be dedicated to that purpose and makes them unavailable to the program application. TI BASIC is designed to be easily readable by humans. Common words, such as PRINT, FOR, NEXT and DATA are used, and statements may read like an ordinary sentence. This way of representing instructions and data to a human being is not necessarily the most efficient way to represent the same information to a machine. A relatively short BASIC program actually generates many times more machine instructions than the number of BASIC statements which comprise it. A single BASIC statement such as

```
10 INPUT X
```

requires many machine instructions to accomplish the stated task. The degree of complexity involved in the execution of a BASIC statement is not apparent to the BASIC programmer. Some programmers feel that BASIC encourages people to generate programs which are not particularly efficient and which may be downright sloppy when compared to a really good assembly language program which accomplishes the same end result. Assembly language is not machine language. It is at a higher level than machine language but is closer to machine language than BASIC. Because it is closer to machine language in its approach and style, it is always possible to write a much more efficient program in assembly language than in BASIC.

Object code takes up far less space both in memory and on a storage device than source code would. A significant reduction in the number of bytes required to store and execute a program frees up more of the computer's resources and power for data storage and manipulation. With the Editor/Assembler module, it is possible to produce object code which is in compressed format. This even further reduces the object file size and storage requirements. As one might expect, any program that is written shorter and simpler will run faster and use less memory.

There are trade-offs, of course. In a high level language such as TI BASIC you can write programs in simple, short, English statements which can tackle some rather complex tasks in a few lines. And, modifying those statements is simple because you do not have to re-assemble your program before running. You may simply type RUN to see the results of the change. With assembly language, while you may not have to code hundreds of instructions, you will have to code far more statements than in

BASIC. You will need to be much more specific and far reaching with your code. With TI BASIC, you were insulated from the actual workings of the computer. You did not need to know anything about how the computer actually did the things you told it. You only had to know how to write the commands in BASIC.

With assembly language programming, you have the computer at your disposal. You are in command. Consequently, you are required to tell the computer far more about each task to be performed. You will need to know a thing or two about the internal details of the computer. The amount of detail in an assembly program will always be greater than an equivalent BASIC program. Most of the subroutines and supporting programs you were accustomed to in TI BASIC and TI Extended BASIC are not there. You must devise and design your own routines as the need arises. This may seem like an inconvenience or nuisance, but actually it is indicative of the degree of power which is handed over to you in assembly language. You have the opportunity to use your creativity in designing any routines you might want. You can customize many of the functions which are provided for you in BASIC. Of course it may be a while before you decide to write an assembly language program which taxes the very limits of your 99/4A, but the potential is there.

Don't be afraid of assembly. It can seem overwhelming at first, but you will learn to appreciate what it can do for you. Because it places more of the power of the computer at your hand, more will be required of your skills as a programmer, but more will be delivered. The payoff comes in speed and performance as well as an increased knowledge and appreciation on your part of how your computer actually accomplishes the tasks you give it.

EDITOR/ASSEMBLER MANUAL REFERENCES

The following references will provide you with some more information on the assembly process.

Section 1.1, page 15
Section 15.1, page 235

Look up these terms in the glossary:

Assembler
Assembling
Assembly Language
Compressed Object Code
Machine Language
Object Code
Syntax
Syntax Definition
TMS9900 Microprocessor

CHAPTER TWO
BINARY AND HEXADECIMAL NOTATION

Before you can begin to program your computer in a low level language such as assembly, it is important to understand how your computer represents and processes information. By knowing how your computer "thinks," you can orient your thinking, and therefore your approach to programming, toward a machine level. Actual machine language is in the form of binary code. Binary refers to the numbering system on which computers are based—the binary, or base two, system. Some understanding of the binary number system is necessary to program in assembly.

Your computer could be thought of as a vast array of miniature switches, each of which may either be on or off. Each of these switches is referred to as a bit. The on/off status of a bit can be used to represent many things: yes or no, high or low, hot or cold, or the values one and zero. If a bit is on, it represents a one, and if the bit is off, it represents a zero. It is, of course, because only two numbers are used (zero and one) that this system is called binary. If you were to consider a series of bits taken as a unit as a value expressed in binary form, large values could be interpreted from the on/off states of a series of bits.

The numbering system you are used to is the decimal system, base ten. When representing a value with a decimal number, the numeric symbols actually represent powers of ten. Thus, the decimal number 2139 can be broken down as follows:

```
  3
1Ø  = 1ØØØ        2 x 1ØØØ  =  2ØØØ
  2
1Ø  =  1ØØ        1 x 1ØØ   =   1ØØ
  1
1Ø  =   1Ø        3 x 1Ø    =    3Ø
  Ø
1Ø  =    1        9 x 1     =     9
                               ----
                               2139
```

The same rules apply to numbering systems with bases other than ten. If you are to deal with significant numbers, you will obviously need more than one bit to do it since a single bit can only represent 1 or 0. A byte is a series of eight bits taken as one unit. Sixteen bits is equal to two bytes, or one word. Here is a byte, represented by eight numbers, each of which can only be a one or a zero:

00001101

In a binary system, each position represents an exponential power of 2. Just as with any other numbering system, leading zeroes have no effect on the value of the expression, so just examine four bits on the right. These are sometimes called the low order bits, or the least significant bits.

```
BIT    VALUE       1       1      Ø      1

PLACE VALUE      EIGHT    FOUR    TWO    ONE

    EXPONENT        3       2      1      Ø
BASE               2       2      2      2
```

The on/off states of these bits represent the value thirteen if taken as a binary number. Here is how you can represent this value as a decimal number:

```
1 x 8 = 8

1 x 4 = 4

Ø x 2 = Ø

1 x 1 = 1
-----------
       13
```

What value does the following byte contain?

00000110

The answer is six. Do you know why? How about this byte?

00001111

If you said fifteen, then you are catching on! How about this byte?

00000001

Well, one is still one, even in binary.

While binary notation applies readily to the on/off states of bits, writing all values in binary format can be quite cumbersome. Needing a shorthand for binary notation, programmers use base 16, or hexadecimal notation. Once again, each digit in hexadecimal notation represents a power of the base, which is sixteen. The exponent values with relation to the position of each hexadecimal numeral would be:

```
PLACE VALUE      4Ø96     256      16     1

    EXPONENT       3        2        1      Ø
BASE              16       16       16     16
```

Throughout your assembly materials from TI, hexadecimal numbers are indicated by the greater – than symbol (">") immediately preceeding the number. Consider the hex number >10. Using the exponent model above, the numeral on the right half of the number represents 0 x 16 or zero. The next numeral would represent 1 x 16 or sixteen. Combining the two answers, 16 + 0 = 16. Thus, >10 = sixteen. In the decimal system, ten symbols (0 – 9) are needed to express values. For the hexadecimal system, sixteen symbols are needed. Digits above 9 are represented by the first six letters of the alphabet (A – F). Thus, "A" represents ten, "B" represents eleven, etc. on up to "F" for fifteen.

The value fifteen in binary would take up four digits: 1111. In hex, the value can be represented in just one digit: F. Thus, a byte of data can be represented in hex with just two digits instead of eight binary digits. This is obviously a more efficient way to express values. Here are some examples:

DECIMAL	BINARY	HEX
1	ØØØØØØØ1	>Ø1
2	ØØØØØØ1Ø	>Ø2
3	ØØØØØØ11	>Ø3
4	ØØØØØ1ØØ	>Ø4
5	ØØØØØ1Ø1	>Ø5
6	ØØØØØ11Ø	>Ø6
7	ØØØØØ111	>Ø7
8	ØØØØ1ØØØ	>Ø8
9	ØØØØ1ØØ1	>Ø9

1Ø	ØØØØ1Ø1Ø	>ØA
11	ØØØØ1Ø11	>ØB
12	ØØØØ11ØØ	>ØC
13	ØØØØ11Ø1	>ØD
14	ØØØØ111Ø	>ØE
15	ØØØØ1111	>ØF
16	ØØØ1ØØØØ	>1Ø
32	ØØ1ØØØØØ	>2Ø
33	ØØ1ØØØØ1	>21

The largest value that can be represented with one byte (eight bits) is binary 1111111, hex >FF, or decimal 255. If you express a word (sixteen bits) as a binary expression, then the largest value of a word is binary 111111111111111, hex >FFFF, or decimal 65,535. Even larger values can be accommodated by using two or more successive words. Whether a value is negative or positive can be of importance for most math and so some way of indicating a number's sign is necessary. The sign of a 16 bit binary expression is indicated by the left bit. If this bit is off (zero), then the value represented by the remaining bits is positive. If that bit is on (one), then the value is negative. The binary number represented by the sixteen bits 0111111111111111 would be +32767. For numbers larger than 32,767 the left bit, or sign bit, would have to be used. To avoid conflict, numbers larger than 32,767 are represented as negative two's complement numbers. Two's complement is a handy way for the computer to deal with binary arithmetic.

Suppose you wanted to compute 16 minus 10. The computer cannot actually subtract, so instead it has to perform two's complement addition. The value to be subtracted is converted to two's complement format and added to the first value. This gives the same answer as subtraction would. Since this is logically the same as negating the second value and adding, the two's complement of a value is said to be negative.

To tackle the problem above, examine the bit values before, after and during the two's complement arithmetic. Since both numbers are small enough, you can use one byte (eight bits) to represent each amount.

	BINARY	DECIMAL	HEX
VALUE #1	ØØØ1ØØØØ	16	>1Ø
VALUE #2	ØØØØ1Ø1Ø	1Ø	>ØA

First, flip all the on bits to off, all the off bits to on for value #2.

```
        ØØØØ1Ø1Ø
becomes 1111Ø1Ø1
```

Notice that the left bit is now on, giving this value a negative sign. Next, add one to the result.

```
1111Ø1Ø1
     + 1
--------
1111Ø11Ø
```

Value #2 is now in two's complement format. Now, add value #1 to value #2.

```
 ØØØ1ØØØØ
+1111Ø11Ø
---------
1ØØØØØ11Ø
```

Disregard the one bit on the left which has been carried over. The remaining bits, 00000110, equal six. Thus 16 − 10 = 6. The computer uses two's complement for negative values and for any value greater than 32,767. This is of importance to you when you wish to write addresses larger than 32,767 in decimal format in programs that access specific addresses. A very easy rule to follow is this: for any address larger than 32,767, subtract 65,536 from the address. For example, to place a value of 79 in address 33008, using the formula 33008 − 65536 = −32528, the TI Extended BASIC language code would look like this:

```
10 CALL LOAD(−32528,79)
```

This most likely will be your major use of two's complement notation, using decimal notation to express addresses larger than 32,767. Two's complement notation does not apply to hex notation. Hex notation handles values greater than 32,767 without any problem.

Go over these examples and practice writing numbers in binary and hex notation. Write your age in each notation. Try any other familiar values. Remember the process is the same for each numbering system. The only difference is the base power each numeral represents.

If all else fails, there are special calculators that do hexadecimal as well as decimal arithmetic and that convert values from one base to another. One is made by Texas Instruments, and it is simply called "The LCD Programmer."

Just like anything else, hex notation comes easier as you do more and more of it. TMS9900 assembly language will allow you to express values in decimal format if you wish. However, because the internal representation of the computer is binary, hex notation most graphically reflects bit values.

Here are four hexadecimal arithmetic problems. See if you can figure them out.

```
1)    >68ØØ     2)   >74Ø2    3)   >DØ66     4)   >ØFAB

    + >Ø29A        - >ØEFØ       + >11ØC        - >ØA95
    -------        -------       -------        -------

            ?              ?             ?              ?
```

Remember, the greater – than symbol is used here to denote that all the numbers are in hexadecimal notation. Since this is a base 16 numbering system, there are few differences between this and decimal arithmetic. The single most confusing thing to beginners of hex are the letters A through F which have taken the place of decimal numbers 10 through 15. Here is an example of simple hex and decimal addition.

```
   HEXADECIMAL          DECIMAL

  >9 + >1 = >A        9 + 1 = 1Ø

  >A + >1 = >B       1Ø + 1 = 11

  >B + >1 = >C       11 + 1 = 12

  >C + >1 = >D       12 + 1 = 13
```

```
>D + >1 = >E          13 + 1 = 14

>E + >1 = >F          14 + 1 = 15

>F + >1 = >1Ø         15 + 1 = 16
```

Note that in the decimal format nine is the number to which you can count before a one must be carried to the left and a zero inserted in the low order digit. In hex, this value is fifteen (F), and a one is carried over to represent sixteen, not ten. When subtracting in hex, sixteen is the value which is borrowed from the left, not ten. First, look at Problem #1. Notice that the numbers are added in columns from right to left, just as in decimal addition.

```
   >68ØØ

 + >Ø29A
 --------
 = >6A9A
    ||||______________  Ø  +  A   =     A
    |||
    |||_______________  Ø  +  9   =    9
    ||
    ||________________  8  +  2   =   A
    |
    |_________________  6  +  Ø   =  6
                                 --------

                                   >6A9A
```

Now, jump to Problem #3. In the right column, six plus twelve equals eighteen, so a one is carried over to the next column to the left. The value of the one carried is actually sixteen, leaving two remaining (eighteen minus sixteen equals two).

```
   >DØ66

 + >11ØC
 ________

   >E172
    ||||______________  6  +  C   =    2
    |||
    |||__ 6 + carry value + Ø  =   7
    ||
    ||________________  Ø  +  1   =  1
    |
    |_________________  D  +  1   = E
                                _______

                                 >E172
```

In Problem #4, subtraction works in a similar fashion.

```
      >ØFFB

    - >ØA95
    --------
    = >Ø566
         |||_______________  B  -  5    =      6
         ||
         ||       eleven - five = six
         ||
         ||
         ||________________  F  -  9    =     6
         |
         |        fifteen -  nine = six
         |
         |
         |_________________  F  -  A    =    5

                  fifteen - ten = five
                                            --------
                                             >Ø566
```

Moving back to Problem #2, examine how the subtraction works. In the second column from the right (zero minus fifteen), a one must be borrowed from the next column to the left. This borrowed value is sixteen, which makes the second column sixteen minus fifteen, which of course is one. After the value is borrowed, the third column from the left becomes three minus fourteen (>3 – >E). Again, a one must be borrowed from the column to the left. This adds sixteen to the value of 3, making it nineteen. Nineteen minus fourteen (>13 – >14) equals five. In the left column, because of the borrowed value, the operation has become six minus zero, which of course is six.

```
    >74Ø2

  - >ØEFØ
  --------
  = >6512
     ||||_______________  2  -  Ø       =      2
     |||
     |||________________  Ø  -  F       =    1
     ||
     ||___ 4 - borrowed value
     |       + carry value - E          =   5
     |
     |___  7 - borrowed value   -  Ø  =  6
                                         -------
                                          >6512
```

With a little practice, you will find that hexadecimal arithmetic is just as easy as decimal. Once you become accustomed to using hex, it will become second nature. It will come in very handy, because this way of representing bytes and words of data tells you a lot about the status of various bits in a short space. Take time now to write down a list of numbers in hex and add and subtract them.

EDITOR/ASSEMBLER MANUAL REFERENCES

The following references will provide you with some more information on binary and hexadecimal notation:

Appendices
Section 24.1 page 393 through Section 24.1.4 page 397

Look up these terms in the glossary:

ASCII
Binary
Bit
Byte
Hexadecimal
Hexadecimal Integer Constant
Nybble
Two's Complement

CHAPTER THREE ADDRESSING

Recall for a moment the conceptual model of the computer as a vast array of switches. This array is analogous to a city street map of buildings, blocks, streets, intersections, and entire communities. To manipulate the bits and bytes of data in the computer, you must "map" the computer's resources by designating certain "communities" and assigning addresses to these areas and to individual bytes within these areas.

Later, you will see various ways to specify addresses in assembly programs. A majority of the program steps you will need to code will involve the movement of data from one area to another or the manipulation of a particular bit, byte, or word. The computer expects you to specify the exact location of the area you wish to access either directly or indirectly. All the possible areas of the computer are numbered to uniquely indentify each one. The number by which the computer locates each byte is its address.

In assembly language programming, a method called base plus displacement addressing is used to calculate and notate internal addresses. Given a known base address, you need only figure the amount of offset, or displacement, needed to arrive at the desired address.

When counting bytes, start with zero as the address of the first byte. Zero is the first positive number to the computer, so always begin counting at zero, not one. Consider a particular area of memory, VDP RAM (Video Display Processor Random Access Memory). In VDP RAM, the first byte (byte zero) represents the first available screen position. In TI BASIC, this would be row 1, column 1. One byte represents one character. For example, if the zero byte of VDP RAM contained the value >41 (decimal 65); the letter "A" would be displayed at row 1, column 1 of the screen. Hex >41 or decimal 65 is the ASCII code for the letter "A." It is by placing the correct values into this area of VDP RAM that symbols and graphics are made to appear on the screen. Most of the functions you will want the computer to do will involve placing certain values into specific areas of the computer.

The area in VDP RAM referred to is the Screen Image Table. There are 768 bytes here which represent all of the available screen positions (24 rows x 32 columns = 768). The corresponding addresses within VDP RAM are 0 through 767 decimal, or >0000 through >02FF hex. In order to make assembly language coding more readable and understandable, addresses do not have to be coded by their numeric values. Instead, you can associate some meaningful name, or label, with the address. Whenever you refer to this label, the assembler will translate it to mean a certain address. This can be done with the EQUate statement.

In TMS9900 assembly language, labels may be up to six characters in length. Establish a label for the first byte of the screen image table. Call the address in VDP RAM where the screen image table begins "SCRTAB." This is an assembly language instruction to do that:

```
SCRTAB EQU >0000.
```

Now you may refer to the first byte as simply SCRTAB, or SCRTAB+0, where +0 represents a displacement value. Adding +0 to SCRTAB would not change the value of the symbolic address SCRTAB. If you wrote a 7 digit number across the top of the screen, it would occupy VDP RAM addresses SCRTAB+0, SCRTAB+1, SCRTAB+2, SCRTAB+3, SCRTAB+4, SCRTAB+5, SCRTAB+6. Notice that in displacement values the first seven bytes of VDP RAM are 0 through 6. To address the last possible screen position (row 24, column 32) you could use the notation SCRTAB+767. Often an address needed must be calculated from some base address and some displacement value. If the number above was 5551234 then the byte values in hex would be:

Symbolic Label Base Address	Displacement Value	Byte Value Hexadecimal	VDP RAM Relative Address	Character Represented by Byte Value
SCRTAB	+ Ø	>35	>ØØØØ	"5"
SCRTAB	+ 1	>35	>ØØØ1	"5"
SCRTAB	+ 2	>35	>ØØØ2	"5"
SCRTAB	+ 3	>31	>ØØØ3	"1"
SCRTAB	+ 4	>32	>ØØØ4	"2"
SCRTAB	+ 5	>33	>ØØØ5	"3"
SCRTAB	+ 6	>34	>ØØØ6	"4"

If you describe to the computer an algorithm for calculating addresses using a base address, you can have a program that can "find its way around" without you having to define in advance any of the internal areas needed. Simply put: NEW ADDRESS = BASE ADDRESS + DISPLACEMENT.

Some diagrams and examples in the TMS9900 assembly reference materials break down individual bytes in order to show the status or importance of each of the eight bits. You will notice that the bits are numbered 0,1,2,3,4,5,6,7. If a full word (two successive bytes, 16 bits) is broken down then the bits are 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F. This is another application of the practice of always counting from zero. Remember, when you are addressing areas of the computer, displacement values count bytes, not bits.

The notation SCRTAB+2 refers to a byte with an address 2 bytes away from the byte SCRTAB. Remember that in assembly language programming, zero is a number that always represents the first of a series when used in the context of addressing or position. If you have done any TI BASIC coding with relative files, you will recall that the first record on a relative file is always the zero record. With the DIM statement for table definition, you may have used the Option Base 0 parameter. This establishes zero as the first susbscript of an array. These are counting schemes similar to base plus displacement.

EDITOR/ASSEMBLER MANUAL REFERENCES

The following references will provide you with more information on addressing.

Appendices
Section 24.2 page 398 through Section 24.2.2 page 402

Look up these terms in the glossary:

Addresses
Addressing Mode
Console
CPU
Memory
RAM
ROM
Symbolic Memory Addressing
VDP RAM

Memory maps can be found in the Editor/Assembler manual and the Mini-Memory manual. Here is another view of the architecture of the TI-99/4A that may help you in visualizing the various memory locations.

```
                              TMS99ØØ CPU MEMORY

>ØØØØ        >2ØØØ        >4ØØØ     >6ØØØ      >8ØØØ      >AØØØ
Console      Memory       Device     Command   CPU PAD     Memory
ROM          Expansion    Service    Module    Memory      Expansion
Operating    "Low" RAM    Routines   ROM/RAM   Mapping     "High" RAM
System                    ROM        Mini-        |        (24K bytes)
GPL                                  Memory       |
Interpreter                                       |
BASIC                                             |
Interpreter                                       |
                                                  |
                                                  |

                                        MEMORY MAPPED PORTS

>8ØØØ    >83ØØ    >84ØØ    >88ØØ    >8CØØ    >9ØØØ    >94ØØ    >98ØØ    >9CØØ
768      256      Sound    VDP      VDP      Speech   Speech   GROM     GROM
Byte     Byte       |      Read     Write    Read     Write    Read     Write
Block    CPU PAD    |        |        |        |        |        |        |
                    |        |        |        |        |        |        |
                 -------  -----------------  -----------  ----------
                 TMS9919  TMS9918A Video     TMS52ØØ      GROM
                 Sound    Display Processor  Speech       ----------
                 Chip                        Synthesizer  >ØØØØ
                            VDP RAM (16K)    -----------  3 Console
                          -----------------  Vocabulary   GROMs
                          >ØØØØ Screen       ROM          containing
                                Image        (32K bytes)  Monitor,
                          >Ø3ØØ Sprite                    Operating
                                Attribute                 System and
                          >Ø38Ø Color                     BASIC
                                Table                     >48ØØ
                          >Ø4ØØ Sprite                    5 Command
                                Descriptor                Module
                          >Ø78Ø Sprite                    GROMs
                                Motion
                          >Ø8ØØ Pattern
                                Generator
                          >1ØØØ Free Space
                                PABs/Buffers
                          >35ØØ Disk DSRs
```

CHAPTER FOUR
REGISTERS

A register is a specially designated word (16 bits, 2 bytes) of storage which has special powers and responsibilities. There are 16 general workspace registers available to the TMS9900 assembly programmer. These general workspace registers are numbered 0 through 15. Registers are the CPU's workspace or scratchpad. They are used for arithmetic, addressing, and bit manipulation. They do special tasks that other available areas of the computer cannot. The general workspace registers always occupy a contiguous area of storage with a total length of >20 (decimal 32) bytes (each register = 16 bits, or 2 bytes; 16 x 2 = 32). Other 32 byte blocks of storage may be designated by your program to be used as general workspace registers as well. Only one set of sixteen may be used at any one time.

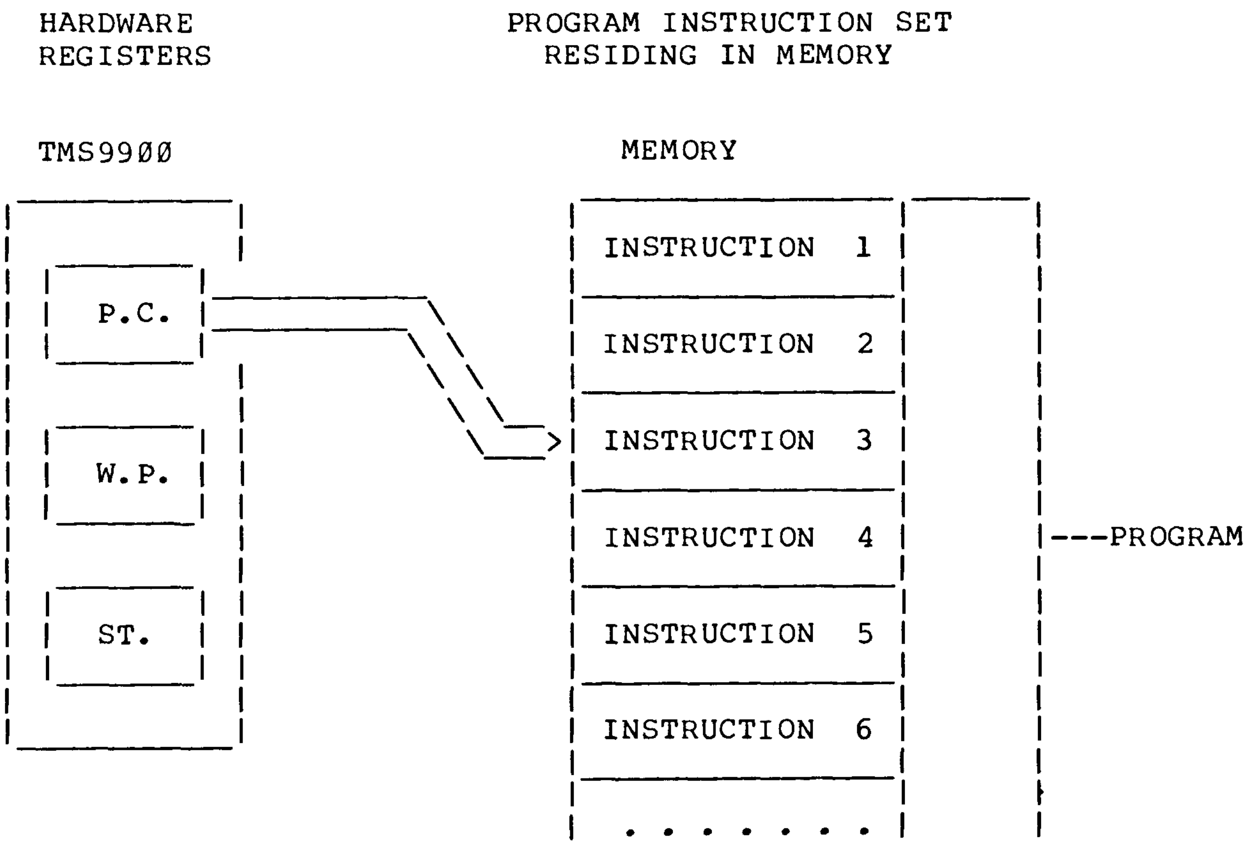
If you are using the Editor/Assembler module, choose option "R" at assembly time. This will automatically label the sixteen general workspace registers R0 through R15. If you are using the Line-by-Line assembler, these symbols are pre-defined. You may also refer to these registers by their numbers (0,1,2,3, etc.). Most people find it far less confusing to stick with the symbols R0,R1,R2,R3,etc.

While registers are special in their uses and functions, their makeup is identical to other storage areas. Each register is a series of 16 bits with on/off states that represent some value in binary format. The hexadecimal notation for the contents of a register is given as four digits (16 bits, 1 hex digit per 4 bits), i.e. >0020.

In addition to the general workspace registers there are three hardware registers used by the computer to manage the computer and the program while it is running. These registers keep track of such things as the address of a subroutine, data or other resources needed by your program, the location of the next instruction to be executed, the resulting status of the last instruction executed, and the beginning address of the general workspace registers. The values contained in these hardware registers will be important to you when designing your assembly language program.

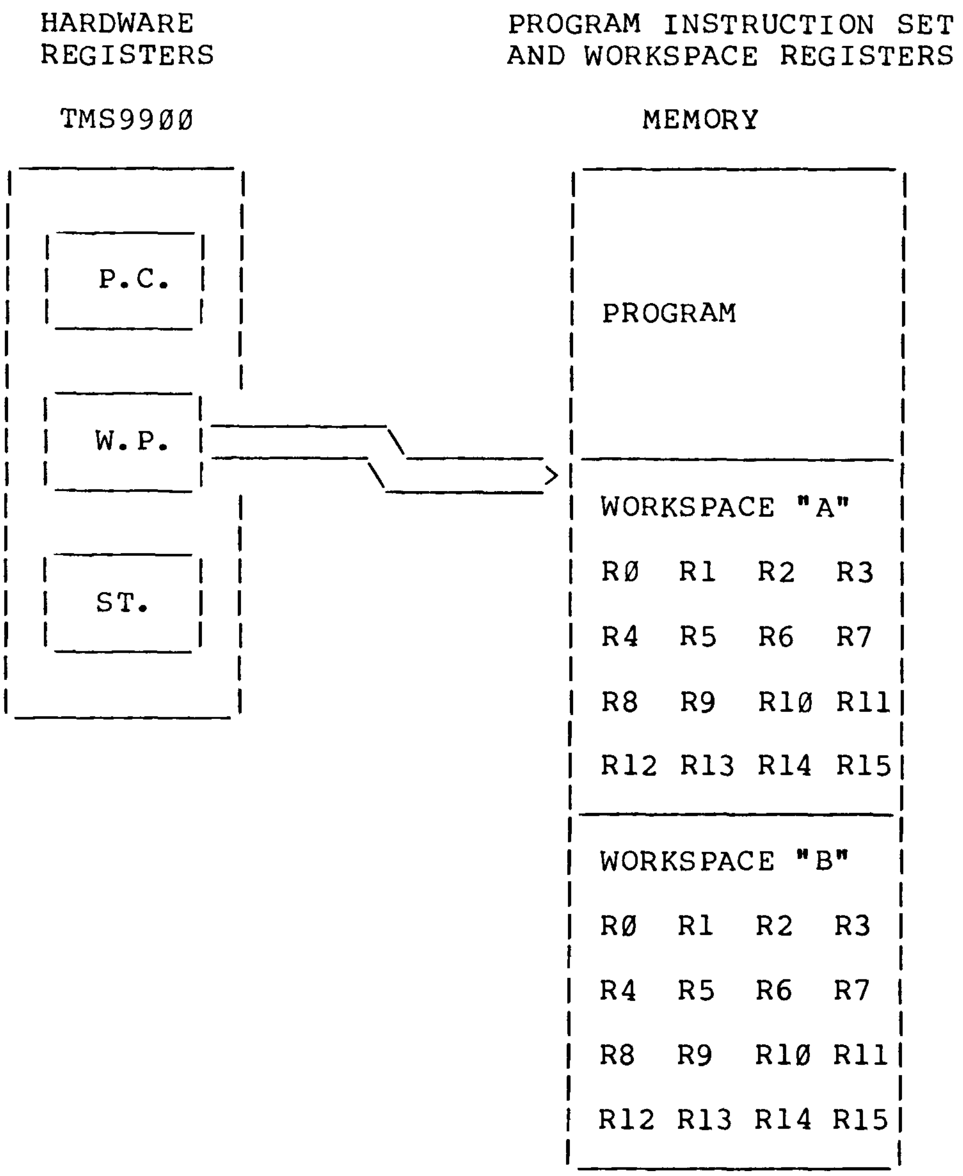
The program counter register (PC) keeps track of a program's instruction set. The values in this register are used in conjunction with other address data to locate or "point to" the next instruction in your program. When your program is run by the computer, all the information contained in it is stored in memory. The address in storage which contains the binary code representing each program instruction is managed by this register. As program instructions are executed, this register is incremented to always point to the address of the next logical instruction.

PROGRAM COUNTER REGISTER (PC)



The workspace pointer register (WP) contains the address pointing to the beginning address of the 16 general workspace registers. Multiple sets of workspace registers can be defined, and each set can be accessed by program manipulation of the address contained in this register. This can be especially useful when subprograms are called by your program which need their own registers.

WORKSPACE POINTER REGISTER (WP)



The status register records the status of the last instruction executed. The individual bits are set (1), or cleared (0) to indicate certain conditions as each instruction executes. Your program code will refer to this register either directly or indirectly quite often. Suppose you wanted to compare value X and value Y. Immediately after the computer executed the compare instruction, the status register would indicate by the status of its bits some relation:

STATUS REGISTER (ST)

BIT POSITION	MEANING IF SET (1)
Ø	LOGICAL GREATER THAN
1	ARITHMETIC GREATER THAN
2	EQUAL
3	CARRY
4	OVERFLOW
5	ODD PARITY
6	EXTENDED OPERATION
7 - 11	NOT USED
12 - 15	INTERRUPT MASK

The status register can let you know if two values are equal or give evidence of many other conditions. Arithmetic operations treat all bytes as representing an arithmetic value. Logical operations treat bytes as representing a series of bits. Various instructions available to you in assembly language will allow you to instruct the computer as to how you wish the contents of a byte to be evaluated. Use of the status register will be covered in other chapters dealing with actual coding.

Some assembly language instructions only apply to registers or are required to involve at least one register. The transfer of data to and from special subprograms is accomplished through certain registers as is the movement of data to and from special areas of the computer. Special addressing calculations and indexing are possible with registers. The three hardware registers and the sixteen general workspace registers do the majority of the "number crunching" and assist in almost every other phase of data handling and management.

Various manufacturers of computers and assembly languages offer differing numbers of workspace registers. Some only have three, some eight, sixteen, or thirty-two. You may never write a program which needs to use all sixteen registers of the TMS9900, but the potential is there. Registers are a component of the microprocessor and are designed into the processor's

Instruction set. It is more efficient and much faster for the microprocessor to operate on its registers than to use a more distant address in memory.

EDITOR/ASSEMBLER MANUAL REFERENCES

The following references will provide you with some more information on registers:

Section 3.1 page 39 through Section 3.1.3 page 40

Look up these terms in the glossary:

Arithmetic Greater Than Bit
Carry Bit
Equal Bit
Logical Greater Than Bit
Odd Parity Bit
Overflow Bit
Program Counter Register
Register
Status Register
Workspace
Workspace Pointer Register

IMPORTANT: Before proceeding to the next section of this book, go to the EDITOR/ASSEMBLER MANUAL REFERENCES at the end of Chapter Five. Read over the recommended pages and glossary terms. Then read the chapter, then read the manual references again as needed.

CHAPTER FIVE
CODING

To put the theory covered so far to use, you will learn to code an assembly program. But first, take time now to read the Editor/Assembler manual references at the end of this chapter. When you have done that, look at this simple program written in TI BASIC. Examine each TI BASIC statement while bearing in mind what has been discussed in the preceding sections.

```
10 CALL CLEAR
20 LET AMTX=10
30 LET AMTY=33
40 LET AMTY=AMTX+AMTY
50 PRINT AMTY
60 END
```

10 CALL CLEAR. This will clear the screen of any characters currently displayed. The TI BASIC program is "calling" a resident TI BASIC subprogram with the name "CLEAR." This routine will do all the things necessary to clear the screen then return control to the next instruction in the TI BASIC program. The next instruction is in line 20.

20 LET AMTX=10. Reserve a storage space in memory to be referred to by the label "AMTX". Then initialize "AMTX" to an arithmetic value of 10.

30 LET AMTY=33. Set aside another storage space. Give this space the name "AMTY". Initialize "AMTY" to an arithmetic value of 33.

40 LET AMTY=AMTX+AMTY. Take the sum found by adding the value at address "AMTX" to the value at address "AMTY" and place it at the address "AMTY".

50 PRINT AMTY. Display the value stored at address "AMTY" on the screen in ASCII symbols. The actual screen location of this display has been predetermined by TI BASIC as the lower left corner of the screen.

60 END. Return control of the computer to the operating system.

As you can see, there are many things that the computer must do to execute each TI BASIC statement. Few of the details about these tasks have been spelled out. The BASIC interpreter and the rest of the "supporting cast" have taken care of all that. In an assembly language program, you will need to be much more exact in telling the computer about its tasks and how it is to perform them.

Try coding an assembly program to do the same thing as the BASIC program above. Each assembly statement has three major parts: a label, an op-code or instruction and one or two operands. Recall the previous assembly statement example:

```
SCRTAB EQU >0000
```

"SCRTAB" is the label. The label starts in the first position of the line and can be up to six characters in length (2 characters with the Line-by-Line assembler), and the first character must be alphabetic. The label is an option and is not always needed. The op-code is separated from the label by at least one space. The op-code is the actual program instruction. The operand field is separated from the op-code by at least one space. There may be one or two operands. If there are two, they must be separated by a comma. The operand field identifies the register or other address that the instruction (op-code) is to operate on. When comments are needed, they may begin at least one space to the right of the last operand. By beginning a new line with an asterisk (*), an entire line can de devoted to comments.

Here is the TMS9900 assembly language code written with the Editor/Assembler package. The "ruler" numbered 1 through 50 represents columns and is included here to illustrate the relative position of each field. (The Mini-Memory example is listed toward the end of this chapter. Instructions for loading and running the program appear in the next chapter.) Look over the following sample program.

```
1-------1Ø--------2Ø--------3Ø--------4Ø--------5Ø

Label  Op-Code Operand(s)

         DEF  START
         REF  VSBW,VMBW
STATUS   EQU  >837C
SAVRTN   DATA >ØØØØ
AMTX     DATA >ØØØA
AMTY     DATA >ØØ21
DECTEN   DATA >ØØØA
HEX3Ø    DATA >ØØ3Ø
PNTANS   BSS  2
WSPREG   BSS  >2Ø
START    LWPI WSPREG
         MOV  R11,@SAVRTN
         BL   @CLEAR
ADDUP    A    @AMTX,@AMTY
         MOV  @AMTY,R5
         CLR  R4
         DIV  @DECTEN,R4
MASKUP   A    @HEX3Ø,R5
         MOV  R5,@PNTANS
         MOV  R4,R5
         CLR  R4
         DIV  @DECTEN,R4
         A    @HEX3Ø,R5
         SLA  R5,8
         MOVB R5,@PNTANS
PUTUP    LI   RØ,738
         LI   R1,PNTANS
         LI   R2,2
         BLWP @VMBW
EOJ      MOV  @SAVRTN,R11
         CLR  RØ
         MOVB RØ,@STATUS
         RT
CLEAR    CLR  RØ
```

```
35          CLR  R1
36 LOOP     BLWP @VSBW
37          CI   RØ,767
38          JEQ  CLEARX
39          INC  RØ
4Ø          JMP  LOOP
41 CLEARX   B    *R11
42          END
```

Do not be alarmed by the sheer number of statements. Assembly languages are always more "wordy" than high level laguages such as BASIC. Each line will be dissected and explained along with many useful commands along the way. Many of the statements in an assembly language program are required entries and show up in program after program. Once you have written several assembly language programs, they become redundant. As you learn to design your own subroutines, they can be used in any other program without having to "go back to the drawing boards" and re-invent them. For all their apparent complexity, all assembly operations are but constant variations on the principles of bits and bytes, addressing, registers, binary arithmetic, and so forth.

The program was written in a simple manner to demonstrate specific program functions with which you are familiar. The program could have been written shorter and "slicker," but then who needs a program just to find the answer to 10+33?! Here is the assembly program:

Line 01 DEF START

DEFine is an assembler directive. A directive is an instruction to the assembler, which is needed for proper assembly of the program. It has no impact on the logical execution of the program. The DEF directive has the effect of placing the name given ("START") into an area in the computer known as the REF/DEF table. Here are kept the names of all programs which are currently in memory. The DEF directive insures that when your program is loaded, its name will be added to the REF/DEF table. When you RUN your program, this is where the RUN software looks for your program name.

"START" is the symbolic address of the point at which program execution commences. The DEF directive must precede the label it defines. The most common practice is to simply make it the first statement in your program. The label used can be any valid label; "START" is just the one used in this program example.

Line 02 REF VSBW,VMBW

The REFerence directive tells the assembler that you intend to use some special resident programs. The REF directive also accesses the REF/DEF table. This directive insures that when your program is loaded these routines will be available to it. VSBW will be equated with the address of the VDP RAM Single Byte Write routine. VMBW will be equated with the address of the VDP RAM Multiple Byte Write routine. These are routines used to display graphics and characters on the the screen. They are just two members of the TMS9900 assembly language "supporting cast."

Line 03 STATUS EQU >837C

This is an EQUate statement. EQUates are also directives. The address is that of the status byte. You will need to refer to the status byte in your program, referring to the actual address by the symbolic label "STATUS." The EQUate directive associates the given label with the actual address of >837C. The actual address along with the symbol "STATUS" is loaded during the assembly process into an area called the symbol table. The assembler uses the symbol table to find the actual address meant whenever you use any symbolic label, such as "STATUS." You must define these relationships between machine addresses and their symbolic names with the EQUate directive.

```
Ø4 SAVRTN    DATA >ØØØØ
Ø5 AMTX      DATA >ØØØA
Ø6 AMTY      DATA >ØØ21
Ø7 DECTEN    DATA >ØØØA
```

These lines use the DATA directive. This directive is used to initialize a word (16 bits, 2 bytes) of memory to some value. If a label is included, that label is associated with the beginning address of the word.

The label represents a symbolic address. The operand contains the value that the word is to be set to. The value may be written in decimal or hexadecimal notation. By using symbolic addresses whenever possible, you do not need to keep track of actual address values. The labels you devise should always be of mnemonic (aiding the memory) value. Your program will be more readable and understandable if the labels picked say something about what they define.

AMTX DATA >000A roughly equals LET AMTX=10
AMTY DATA >0021 roughly equals LET AMTY=33

A similar directive is the BYTE directive. The statement MYBYTE BYTE >04, intitializes one byte (8 bits) of memory. The effect of DATA and BYTE are similar; the only difference is the number of bits which are initialized (8 vs. 16). The Line-by-Line assembler does not recognize the BYTE directive.

```
Ø9  PNTANS  BSS  2
1Ø  WSPREG  BSS  >2Ø
```

Lines 9 and 10 use the Block Starting with Symbol (BSS) directive. This reserves blocks of memory without any initialization. These areas will be used as workspace by the program and will be part of the program. In line 9, two bytes of memory have been reserved. They will be referred to as "PNTANS." Line 10 sets aside 32 (>20) bytes called "WSPREG."

The first quarter of this program has been covered, and no instruction has been performed. All that has been done so far might be thought of as housekeeping, things that must be done in preparation. When coding TI BASIC statements, you could define variables in the same statement in which they were first used. Not so in an assembly language. You must define all labels and workspace areas before you can refer to them in any program statement. Deciding your program's housekeeping needs means planning and forethought.

Line 11 START MOV R11,@SAVRTN

Here is the label "START" that was defined in line 1. The first order of business is to save the entry address of the program. Register 11 is the computer's general purpose linkage (addressing) register. When your assembly program begins execution, the address to which your program should return when done is in R11. This address is vital to successful program completion. You want to save that address since R11 will be used elsewhere in the program. The word of memory called SAVRTN was set aside just for this purpose. MOVe the value in R11 to the storage location. The address is a symbolic address, which is represented in a MOVe instruction by the "at" symbol (@). The MOV operation copies a word (16 bits, 2 bytes) of a register or other address in memory to another register or address. The sending storage location remains unchanged, while the receiving storage location becomes its clone. Suppose R11 contains the address value >3238. Before the MOV instruction:

```
  R11      @SAVRTN

 >3238     >ØØØØ
```

After the MOV:

```
  R11      @SAVRTN

 >3238     >3238
```

Note that the entire word (16 bits, 2 bytes) is affected. Suppose that you only needed to move one byte (8 bits) at a time. Then the MOVB (Move Byte) instruction can be used. Throughout TMS9900 assembly language you will find parallel instructions which address either one word (16 bits) at a time or one byte (8 bits) at a time. If you use a byte instruction with a register or other full word address, the instruction always uses the left byte (high order byte). For example:

```
    MOVB  R3,R4
```

Suppose that the contents of the registers involved before the instruction was executed were:

```
  R3        R4

 >1Ø4C     >ØØ11
```

Then after the MOVB:

```
  R3        R4

 >1Ø4C     >1Ø11
```

Notice that the right byte (low order byte or least significant byte) is unaffected in either register by the MOVB instruction.

Line 12 LWPI WSPREG

Now you Load Workspace Pointer Immediate (LWPI). You need to establish an alternate area of workspace registers for use by special routines that are needed. They can have their own set of general workspace registers. The effect of this statement is to point to the address of the block of memory that was defined in line 10. This is a statement you will typically need in any stand-alone TMS9900 assembly language program.

Line 13 BL @CLEAR

Instead of calling a resident routine to clear the screen, this program has its own routine coded. The Branch and Link (BL) instruction is roughly equivalent to the GOSUB with RETURN you have seen in TI BASIC. Control is passed from this point in the program to the address CLEAR, and once again the return address (address of the next sequential instruction) is loaded into R11. When coding TI BASIC statements, you had to GOTO line numbers, and sometimes resequencing could be disastrous if you had an unresolved line number. One of the niceties of using labels is that line numbers have no affect on program logic. CLEAR refers to the beginning address of the CLEAR subroutine no matter what line number it is on. R11 now contains the address of line 14. That is where you want to return to when you have finished the CLEAR routine. Program execution now transfers to:

```
Line  34     CLEAR    CLR   RØ
      35              CLR   R1
```

"CLEAR" is the label to which you have instructed the computer to branch. The first step of the CLEAR routine is to set all the bits in registers R0 and R1 to zeroes. The CLR instruction clears (sets the bits to all zeroes) a word of memory or a register at a time. R0 and R1 now contain:

```
  RØ       R1

 >ØØØØ    >ØØØØ
```

```
Line  36     LOOP   BLWP  @VSBW
```

Now you can begin to "fiddle with" the video display processor chip. The label LOOP will be used to build a simple loop much like a FOR-NEXT loop in TI BASIC. A loop allows you to execute one instruction many times.

The instruction BLWP is like the branch and link except that this time you want the workspace pointer register (WP) to point to the alternate workspace registers established at line 11. This is required of the VSBW and other resident routines. BLWP stands for Branch and Load Workspace Pointer. The address to which program execution branches is the address of the VDP RAM Single Byte Write routine. You can pass values to this program via R0 and R1. Into R0 you put the destination address in VDP RAM to which you want to write. Place the single byte of data into the left byte of R1.

Currently R0 and R1 contain all zeroes. Remember that VDP RAM address zero corresponds to row 1, column 1 of the screen and R0 addresses VDP RAM. The VSBW routine has written to this address the left byte (8 bits) of R1 (all zeroes). What will be displayed at row 1, column 1? Nothing! Then, if you add 1 to the value in R0 and repeat this step, VDP RAM address 01 (row 1, column 2) is cleared. There are 768 screen positions to be cleared. These VDP RAM addresses are 0 through 767 decimal, or >0000 through >02FF hex. Thus, by looping through these steps until R0 has been incremented to a value of 767, the entire screen can be cleared. The next statement checks R0 for this value.

```
Line  37          CI   RØ,767
      38          JEQ  CLEARX
```

Line 37 is a Compare Immediate (CI) instruction. This is used to compare the value of a register to a known value. As a result, the bits in the status register are affected and are tested by the next line. In addition to comparing registers to known values, there are compare instructions for word to word comparisons and byte to byte comparisons.

The Jump if EQual (JEQ) instruction completes the comparison by directing some action based upon the result. This instruction checks the status register's equal bit and, if it is set, transfers control to the label CLEARX. Jump instructions are like short range branch or GOTO instructions. The addresses they use must be within 256 bytes of the instruction itself. If the difference is too great, the error "Out Of Range" appears during assembly. Jump instructions do not need the "@" prefix on symbolic addresses. The first time through this loop, the equal condition is not true and the JEQ instruction at line 38 has no effect on program execution.

Line 39 INC R0

INCrement the value in R0 by binary 1. Remember this value is being used as an address in VDP RAM. Each time through, you increment that address by 1. The INC instruction will increment (add a binary 1 to) the register or word of memory specified in the OPERAND. There is also an INCT instruction which will increment by 2.

```
Line  4Ø         JMP    LOOP
      41  CLEARX B      *R11
```

Line 40 is an unconditional jump to the label LOOP which completes the loop described above. When a value of 767 is reached in R0, program execution transfers to line 41, CLEARX. This instruction is an unconditional branch (like a GOTO) to the address in R11. The use of the asterisk (*) immediately preceding the named register indicates that the value in R11 is to be used as an address. The BL instruction at line 13 put the address of line 14 into R11 before the subroutine CLEAR was performed. You are now instructing the computer to branch to the address in R11, which is the address of line 14.

Line 14 ADDUP A @AMTX,@AMTY

The label ADDUP helps you remember what this step does. The contents of the word (16 bits, 2 bytes) at symbolic address AMTX is added to the value at address AMTY. Both addresses require the @ prefix for this step.

```
Before the Add:     @AMTX          @AMTY

                    >ØØØA          >ØØ21        33

                                 + >ØØØA      + 1Ø
                                 -------      ----
After the Add:      >ØØØA          >ØØ2B     =  43
```

Now the answer (43) is at AMTY. But the value in AMTY is a binary value, not the correct ASCII code for representing the characters "43" on the screen. You need to display a >34 (the ASCII code for the symbol "4") at one screen position and >33 (ASCII for "3") at the very next screen position in order to display "43". The next series of instructions will convert the answer to its displayable format.

```
Line  15          MOV   @AMTY,R5
      16          CLR   R4
```

Registers 4 and 5 will be used for the arithmetic needed. The answer (still in binary format) is moved to R5 and R4 is cleared.

Line 17   DIV @DECTEN,R4

The DIVide instruction does just that, divide. DIV uses two successive registers, in this case R4 and R5. You only need to specify R4 in the second operand since the use of the next available register (R5) is implied. The first operand @DECTEN is the divisor. This statement will divide the value in R5 by the value at DECTEN and put the answer in R4 and any remainder into R5. Before the DIV:

	@DECTEN	R4	R5
	>ØØØA	>ØØØØ	>ØØ2B
After the DIV:	>ØØØA	>ØØØ4	>ØØØ3

In decimal, this would be the same as 43 divided by 10 equals 4 with a remainder of 3.

Line 18 MASKUP A @HEX30,R5

This Adds the value at HEX30 to the value in R5 and puts the answer into R5. Before line 18 is executed, R5 contains >0003, a binary three. The ASCII code for a displayable "3" is >33. The difference between this and the answer is >30 (>33 minus >03 = >30). This >30 or HEX30 "mask" must be added to the binary value to make it a proper ASCII numeral. Before the Add:

	@HEX3Ø	R5
	>ØØ3Ø	>ØØØ3
After the Add:	>ØØ3Ø	>ØØ33

R5 now contains the ASCII code for "3". This is the first digit of your displayable answer.

Line 19   MOV R5,@PNTANS

Save this much of the answer in the area which was set aside at line 9, PNTANS. This is a full word move, 16 bits. Before the MOV:

	R5	@PNTANS
	>ØØ33	>ØØØØ
After the MOV:	>ØØ33	>ØØ33

```
Line  2Ø          MOV   R4,R5
      21          CLR   R4
      22          DIV   @DECTEN,R4
      23          A     @HEX3Ø,R5
```

In lines 20 through 23 the process is repeated for the second digit of the answer. When 43 was divided by 10 at line 17, the answer 4 was placed in R4. The answer of 4 is divided again by 10. To do that, you need to place it in R5 and set R4 to all zeroes. Four divided by 10 yields an answer of zero with a remainder of 4. The remainder is placed into R5 to which a >30 mask is added. In this type of conversion logic, you are operating on the remainder in R5. R5 now contains >0034 (the ASCII code for "4").

```
Line  24          SLA   R5,8
      25          MOVB  R5,@PNTANS
```

Now you have the "4" portion of the "43" you wish to display. The next step is to move it to PNTANS and match it up to the "3." Since you do not want to destroy the "3" now at PNTANS, a byte-sized move would be better than a word-sized move. Remember that byte instructions always operate on the left byte. R5 contains >0034, which means that the value to move is in the "wrong" byte. (There are many approaches to all this and the method used here is contrived to illustrate instruction usage). One way to approach this is shown in line 24. This is a SHIFT instruction and is one of the special things that only registers can do. The particular instruction used is the Shift Left Arithmetic (SLA) instruction. Line 24 specifies that the bits in register five are to be shifted to the left 8 positions and that the right side of the register be filled with zeroes Here are the contents of R5 broken down into individual bits shown before and after the SLA is performed:

```
Before the SLA:           REGISTER FIVE - BINARY               HEX

                 BIT      Ø 1 2 3 4 5 6 7 8 9 A B C D E F
                          - - - - - - - - - - - - - - - -
                 VALUE    Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø 1 1 Ø 1 Ø Ø   >ØØ34
                          - - - - - - - - - - - - - - - -
                                 <------ SHIFT 8 PLACES

After the SLA:            - - - - - - - - - - - - - - - -
                          Ø Ø 1 1 Ø 1 Ø Ø Ø Ø Ø Ø Ø Ø Ø Ø   >34ØØ
                          - - - - - - - - - - - - - - - -

Before the MOVB:   R5           @PNTANS

                  >34ØØ         >ØØ33

After the MOVB:   >34ØØ         >3433
```

```
Line  26   PUTUP   LI   RØ,738
      27           LI   R1,PNTANS
      28           LI   R2,2
      29           BLWP @VMBW
```

Lines 26 through 29 use the VDP RAM Multiple Byte Write (VMBW) subroutine to display the final answer. Lines 26, 27, 28 use the Load Immediate (LI) instruction. LI is used to place values into registers. Like the Compare Immediate instruction, LI is used with specific values. The first operand names a register, the second, the value. R0 is loaded with the VDP RAM address of the desired screen position. R1 is loaded with the address (symbolic or real) of the data to be moved (not the data itself, but the beginning address of where the data is to be found.) Into R2 is loaded the length in bytes of the data to be moved. The branch to the subroutine places >3433, found at address PNTANS, at VDP RAM address 738 (lower left corner of the screen). VDP RAM address 738 contains >34 (ASCII for "4") and VDP RAM address 739 contains >33 (ASCII for "3").

```
Line    3Ø   EOJ    MOV    @SAVRTN,R11
        31          CLR    @STATUS
        32          DECT   R11
        33          RT
```

These lines complete the program. Line 42, the END directive, is a required entry and instructs the assembler that this is the end of the source code. If the label START were included as an operand of the END directive (and you are using the Editor/Assembler package):

42 END START

then this program would begin to execute as soon as it is loaded. Depending on the application for which an assembly program is written, this may, or may not, be suitable.

In line 30 the return address which was saved in @SAVRTN is MOVed to R11. Line 31 CLeaRs (sets to all zeroes) the word of memory at address >837C. Address >837C is the address of the GPL (Graphics Programming Language) status byte. You are in effect telling the computer's operating system that everything is okay by clearing this location.

Line 32 DECrements by Two the value in R11. DEC and DECT are just the opposite of INC and INCT. They subtract by 1 and 2 respectively. By branching to an address which is 2 less than the entry point address, the computer will "freeze" the displayed answer and will not do anything else until Quit (FCTN =) is pressed. Altering the return address in this way is not really the proper way to end your program. At some point, try this program without the DECT R11 instruction at line 31, and you'll see how incredibly fast TMS9900 assembly language is. The screen will barely blink as the program runs and ends.

Line 33 uses the RT instruction. This instruction has the same effect as "B *R11". The two are interchangeable. The return (RT) instruction simply makes the code more understandable and readable. It has more mnemonic (aiding the memory) value than "B *R11" does. The program performs an unconditional branch to the address in R11. This completes program execution and returns control of the computer to the operating system.

Here is the assembly listing produced with the Editor/Assembler package. Options used were "R" (label general workspace registers R0-R15), "L" (produce a listing), "S" (print the symbol table), and "C" (produce object code which is in compressed format).

```
 99/4 ASSEMBLER
VERSION 1.2                                         PAGE ØØØ1

  ØØØ1                          DEF   START
  ØØØ2                          REF   VSBW,VMBW
  ØØØ3       837C   STATUS      EQU   >837C
  ØØØ4  ØØØØ ØØØØ   SAVRTN      DATA  >ØØØØ
  ØØØ5  ØØØ2 ØØØA   AMTX        DATA  >ØØØA
  ØØØ6  ØØØ4 ØØ21   AMTY        DATA  >ØØ21
  ØØØ7  ØØØ6 ØØØA   DECTEN      DATA  >ØØØA
  ØØØ8  ØØØ8 ØØ3Ø   HEX3Ø       DATA  >ØØ3Ø
  ØØØ9  ØØØA        PNTANS      BSS   2
  ØØ1Ø  ØØØC        WSPREG      BSS   >2Ø
  ØØ11  ØØ2C C8ØB   START       MOV   R11,@SAVRTN
        ØØ2E ØØØØ'
  ØØ12  ØØ3Ø Ø2EØ               LWPI  WSPREG
        ØØ32 ØØØC'
  ØØ13  ØØ34 Ø6AØ               BL    @CLEAR
        ØØ36 ØØ7E'
  ØØ14  ØØ38 A82Ø   ADDUP       A     @AMTX,@AMTY
        ØØ3A ØØØ2'
        ØØ3C ØØØ4'
  ØØ15  ØØ3E C16Ø               MOV   @AMTY,R5
        ØØ4Ø ØØØ4'
  ØØ16  ØØ42 Ø4C4               CLR   R4
  ØØ17  ØØ44 3D2Ø               DIV   @DECTEN,R4
        ØØ46 ØØØ6'
  ØØ18  ØØ48 A16Ø    MASKUP     A     @HEX3Ø,R5
        ØØ4A ØØØ8'
  ØØ19  ØØ4C C8Ø5               MOV   R5,@PNTANS
        ØØ4E ØØØA'
  ØØ2Ø  ØØ5Ø C144               MOV   R4,R5
  ØØ21  ØØ52 Ø4C4               CLR   R4
  ØØ22  ØØ54 3D2Ø               DIV   @DECTEN,R4
  ØØ23  ØØ58 A16Ø               A     @HEX3Ø,R5
        ØØ5A ØØØ8'
  ØØ24  ØØ5C ØA85               SLA   R5,8
  ØØ25  ØØ5E D8Ø5               MOVB  R5,@PNTANS
        ØØ6Ø ØØØA'
  ØØ26  ØØ62 Ø2ØØ   PUTUP       LI    RØ,738
        ØØ64 Ø2E2
  ØØ27  ØØ66 Ø2Ø1               LI    R1,PNTANS
        ØØ68 ØØØA'
  ØØ28  ØØ6A Ø2Ø2               LI    R2,2
        ØØ6C ØØØ2
  ØØ29  ØØ6E Ø42Ø               BLWP  @VMBW
        ØØ7Ø ØØØØ
  ØØ3Ø  ØØ72 C2EØ   EOJ         MOV   @SAVRTN,R11
        ØØ74 ØØØØ'
  ØØ31  ØØ76 Ø4EØ               CLR   @STATUS
        ØØ78 837C
  ØØ32 ØØ7A Ø64B              DECT R11
  ØØ33 ØØ7C Ø45B              RT
  ØØ34 ØØ7E Ø4CØ  CLEAR       CLR  RØ
  ØØ35 ØØ8Ø Ø4C1              CLR  R1
  ØØ36 ØØ82 Ø42Ø  LOOP        BLWP @VSBW
       ØØ84 ØØØØ
  ØØ37 ØØ86 Ø28Ø              CI   RØ,767
       ØØ88 Ø2FF
  ØØ38 ØØ8A 13Ø2              JEQ  CLEARX
  ØØ39 ØØ8C Ø58Ø              INC  RØ

 99/4 ASSEMBLER
VERSION 1.2                                     PAGE ØØØ2
  ØØ4Ø ØØ8E 1ØF9              JMP  LOOP
  ØØ41 ØØ9Ø Ø45B  CLEARX      B    *R11
  ØØ42                        END
```

The first column of numbers are your line numbers. The values in the second column (0000, 0002, 0004, etc.) represent the location counter. With the Editor/Assembler, it starts at >0000 and is incremented to address each line. The location counter goes from >000A to >000C at line 10 because the BSS directive at line 9 sets aside 2 bytes of storage that are part of the program. So, >000A + 2 = >000C. Notice that the first three directives do not affect the location counter values. The values seen here are displacement values. They are added to the beginning address where the code is loaded in order to address each line or label in the program.

The third column of numbers are hex representations of the machine code at each address (location counter value). At line 5, location >0002, you will see the value >000A (ten) which is the value to which AMTX is initialized. Line 16, location >0042, shows the value >04C4. This is hex for the machine language instruction to clear (CLR). The length of your program can be determined by subtracting the beginning value of the location counter from the last value in the location counter. In this case >0090 - >0000 = >0090, indicating that this program is >90 (144 decimal) bytes long.

Here is the symbol table that was built from the program. Each symbol which was used along with its address is shown in alphabetic order. The address next to each symbol may be an actual address (STATUS >837C), or the value of the location counter.

```
 99/4 ASSEMBLER
VERSION 1.2                                              PAGE ØØØ3

' ADDUP  ØØ38  ' AMTX   ØØØ2  ' AMTY   ØØØ4  ' CLEAR  ØØ7E
' CLEARX ØØ9Ø  ' DECTEN ØØØ6  ' EOJ    ØØ72  ' HEX3Ø  ØØØ8
' LOOP   ØØ82  ' MASKUP ØØ48  ' PNTANS ØØØA  ' PUTUP  ØØ62
  RØ     ØØØØ    R1     ØØØ1    R1Ø    ØØØA    R11    ØØØB
  R12    ØØØC    R13    ØØØD    R14    ØØØE    R15    ØØØF
  R2     ØØØ2    R3     ØØØ3    R4     ØØØ4    R5     ØØØ5
  R6     ØØØ6    R7     ØØØ7    R8     ØØØ8    R9     ØØØ9
' SAVRTN ØØØØ  D START  ØØ2C    STATUS 837C  E VMBW   ØØ7Ø
E VSBW   ØØ84  ' WSPREG ØØØC

  ØØØØ ERRORS
```

LINE-BY-LINE ASSEMBLER/MINI-MEMORY NOTES

To accomplish the same results in an assembly language program using the Line-by-Line assembler, several things must be done differently. The Line-by-Line assembler does not recognize some of the directives that the Editor/Assembler does. Some instructions are needed with the Line-by-Line assembler program which are not usually needed with the Editor/Assembler.

When your object program is run, all the information in it is loaded into memory. With the Editor/Assembler package and a program as simple as this one, the actual address at which the program is stored is of little importance. Simply allow the loader to start loading the program wherever it wants. Except for some special programs this is almost always the case. With the Mini-Memory however, your assembly language programs must be loaded into a limited amount of space. Within this same space you must allow for the symbol table's storage requirements as well as the REF/DEF table. The symbol table is built during assembly from symbols you use in your program. The REF/DEF table is an area where the computer stores the program name and the names of other routines (VSBW, VMBW, etc).

The default load address with the Line-by-Line assembler is >7D00. If you do not use a command which alters the location counter, then >7D00 is the value it will start at when using the Line-by-Line assembler. The symbol table built from your program begins at address >7CD8 and contains one 4 byte entry for each symbol you have used and one additional 4 byte entry which marks the end of the table. If you generate too many symbols in your assembly program, the default load address of >7D00 will not be suitable. Your symbol table will have grown to exceed this address and your program , when loaded, will overlay this table. For this reason, it is wise to limit the use of EQUates and labels (which generate symbol table entries) when using the Line-by-Line assembler. To determine the optimum load address for your assembly language program using the Line-by-Line assembler, use this formula:

Load address = >7CD8 + (4 * (number-of-labels + 1))

For example, if there are 12 labels:

```
4 * (12+1) = 4 * 13 = 52 or >34

   >7CD8
 + >ØØ34
 --------
 = >7DØC
```

You now know that the optimum load address is >7D0C, not >7D00. In order to get the value loaded into your program, you must include a directive known as AORG (Absolute ORiGin).This would be coded as:

```
        AORG >7D0C
```

AORG is the op-code. There is no label. This instruction affects the the location counter. The value in the location counter is where your assembled code is stored. In the above example, the AORG directive should be the first or nearly the first statement in your program. However, it can be used again and again. If you had a large program, you might want to start out loading the first section of it at one address and continue until that space was used up. Then by placing an AORG statement in the program before the next section of code, the remaining portion of the program could be stored beginning at a new location. This kind of application is only feasible when the 32K memory expansion is attached and represents an advanced programming technique. With the Line-by-Line assembler, use fewer labels. Use specific values instead. Perform the calculation above to determine your optimum load address.

Now that you know something about AORG, look over the following program example. Here is the same program as it would have to be written to be entered using the Line-by-Line assembler:

```
Line          Op
Number  Label Code    Operand
   |     |      |      |
   |     |      |      |
------------------------------
   ØØ1          AORG   >7DØC
   ØØ2  VS      EQU    >6Ø24
   ØØ3  VM      EQU    >6Ø28
   ØØ4  AX      DATA   >ØØØA
   ØØ5  AY      DATA   >ØØ21
   ØØ6  DT      DATA   >ØØØA
   ØØ7  H3      DATA   >ØØ3Ø
   ØØ8  PA      BSS    2
   ØØ9  WR      BSS    >2Ø
   Ø1Ø  ST      MOV    11,6
   Ø11          LWPI   WR
   Ø12          BL     @CL
   Ø13          A      @AX,@AY
   Ø14          MOV    @AY,5
   Ø15          CLR    4
   Ø16          DIV    @DT,4
   Ø17          A      @H3,5
   Ø18          MOV    5,@PA
   Ø19          MOV    4,5
   Ø2Ø          CLR    4
   Ø21          DIV    @DT,4
   Ø22          A      @H3,5
   Ø23          SLA    5,8
   Ø24          MOVB   5,@PA
   Ø25          LI     Ø,738
   Ø26          LI     1,PA
   Ø27          LI     2,2
   Ø28          BLWP   @VM
   Ø29          CLR    @>837C
   Ø3Ø          DECT   6
   Ø31          B      *6
   Ø32  CL      CLR    Ø
   Ø33          CLR    1
   Ø34  LP      BLWP   @VS
   Ø35          CI     Ø,767
   Ø36          JEQ    CX
   Ø37          INC    Ø
   Ø38          JMP    LP
```

```
Ø39 CX     B      *11
Ø4Ø        AORG   >7Ø1C      *These lines show how to add program
Ø41        DATA   >7FB2      *name and entry point to REF/DEF
Ø42        DATA   >7FEØ      *table. Though this is not included
Ø43        AORG   >7FEØ      *in subsequent program listings,
Ø44        TEXT   'EXAMP1'   *this step is required with any
Ø45        DATA   ST         *program entered with Line-by-Line
Ø46        END               *assembler.
```

Labels used with the Line-by-Line assembler can only be 2 characters in length. Whenever a label is used, it will generate a symbol table entry. With the space restrictions of the Line-by-Line assembler, it is important to limit the use of labels. For this reason, most of the labels shown in the Editor/Assembler version have been eliminated.

The Line-by-Line assembler does not recognize the directives DEF and REF. An alternate method of adding your program's name to the REF/DEF table will be discusssed in the next chapter. In the Editor/Assembler example, the REF directive was used to automatically equate the symbolic names VSBW and VMBW with their addresses. In the Line-by-Line assembler example, this must be accomplished with EQUates at lines 2 and 3. The names given these routines have been shortened to the 2 character limit as well.

The symbolic addresses SAVRTN and STATUS have been eliminated order to conserve label usage and therefore symbol table entries. Instead, at line 10, the contents of register 11 (the return address) is moved to register 6. The "R" prefixes have been left off of all references to registers in this program example for two reasons. First, R0,R1,R2,R3, etc. will result in symbol table entries, so this is one way to cut down. Second, you may be able to decide for yourself from this example whether or not this method of register naming is for you. Most people find the symbols R0,R1,R2,etc. easier to follow. Register 6 will not be used anywhere else in this program, so it will do just fine for temporary storage.

At line 29, you want to clear the status byte just as you did in the Editor/Assembler version. Simply code the specific address as @>837C instead of the symbol STATUS, which was dropped. Both ways of coding are valid. In the Editor/Assembler version you have certain "luxuries" and don't have to be as conscious of symbols. You could have coded the specific address in the Editor/Assembler version just as you could have used a symbolic address in the Line-by-Line assembler version.

The RT instruction is not recognized by the Line-by-Line assembler either. Instead, use an unconditional branch to the return address in register 6. Your return address was moved back to register 11 in the Editor/Assembler version because RT uses register 11.

Lines 40 through 45 are needed to add the program name (EXAMP1) and the entry point (ST) to the REF/DEF table. There will be more on this in the next chapter. Line 44 uses the TEXT directive. The TEXT directive allows you to initialize storage much the same as the DATA directive. With the TEXT directive the values may be entered as a string of characters enclosed by single quotation marks. Each character takes up one byte of storage.

There are quite a few of these coding differences between the Line-by-Line assembler and the Editor/Assembler. Generally they are the directives recognized, label lengths, program length, and the assembler/loader. The instruction set and the way they operate are practically indentical. The explanations are valid for either assembler product.

EDITOR/ASSEMBLER MANUAL REFERENCES

The following references will provide you with more information on coding.

Section 3.3 page 46 through Section 3.9 page 55
Section 4.1 page 56 through Section 5.8 page 74

ADDRESSING—There are five general addressing modes. Three are used in the program example:

```
                                         Example
Workspace Register Direct              MOV 11,6
Workspace Register Indirect            B   *R11
Symbolic Memory                        A   @AMTX,@AMTY
```

Section 14.1 page 208 through Section 14.1.1 page 210
Section 14.1.4 page 212
Section 14.3 page 224 through Section 14.4.2 page 228
Section 14.5.2 page 234

DIRECTIVES—These directives are covered in the program example:

AORG - Absolute origin, affecting the location counter.
BSS - Block Starting with Symbol, reserves storage.
EQU - Equate, associates a label with an address.
BYTE - Initialize 8 bits of storage to some value. Not recognized by Line-by-Line Assembler.
DATA - Initialize 16 bits of storage to some value.
TEXT - Initialize storage using a character string.
DEF - Define, makes the labels it defines part of your object code so that your program is available to other software and adds the labels to the REF/DEF table. Not recognized by the Line-by-Line assembler.
REF - External Reference, make other program labels available to your program. Not recognized by the Line-by-Line assembler.
END - End of source code.

Section 6.1 page 78 through Section 6.14.2 page 102

ARITHMETIC INSTRUCTIONS—These arithmetic operations are used and detailed in the program example:

A - Add words, adds the operands and puts the answer at the second operand.
INC - Increment the contents of the operand by 1.
INCT - Increment the contents of the operand by 2.
DEC - Decrement the contents of the operand by 1.
DECT - Decrement the contents of the operand by 2.
DIV - Divide, divides the first operand into the second operand (which must name a register).

Each arithmetic instruction affects the status register according to the result of the operation.

JUMP AND BRANCH INSTRUCTIONS—Five of the many kinds of jump and branch instructions used in the example are:

B - Unconditional Branch, branch to a secified address.
BL - Branch and Link, branch and place the return address into R11.
BLWP - Branch and Load Workspace Pointer, branch to a routine and set the WP register to point to that routine's own register space.
JMP - Jump, unconditional jump to an address.
JEQ - Jump if EQual, if the equal bit in the status register is 1, then jump to the specified address.

Section 8.1 page 138 through Section 8.3 page 143.

COMPARE INSTRUCTIONS—Only one type of compare instruction is used in the program example. However, they all operate in similar fashion.

CI - Compare Immediate, compare the contents of the named register (first operand), to a specific value (second operand).

Section 10.1 page 161 through Section 10.5 page 168.
Section 10.9 page 172

LOAD AND MOVE INSTRUCTIONS—There are eight of these instruction types. The example program used four:

LWPI - Load Workspace Pointer Immediate, needed to establish alternate workspace registers.
MOV - Move a word, copies a word (16 bits) to another word.
MOVB - Move Byte, copies one byte (8 bits) to another byte.
LI - Load Immediate, used to put specific values into a register.

These instructions affect the status register according to the value at the address involved.

Section 11.5 page 184.

LOGICAL INSTRUCTIONS—Only one is used in the example program:

CLR - Clear, set all the bits at the named register or address to all zeroes.

Section 12.1 page 194 through Section 12.5 page 204.

WORKSPACE REGISTER SHIFT INSTRUCTIONS—Again, only one is used in the program example, but its operation is analogous to the other types of register shift instructions.

SLA - Shift Left Arithmetic, shift the bits of the named workspace register a specified number of positions to the left and fill the right side with zeroes. All shift instructions affect the status register.

Section 13.1 page 206 through Section 13.2 page 207.

PSEUDO INSTRUCTIONS—Only one is used and it is not recognized by the Line-by-Line assembler.

RT - Return, has the same effect as "B *R11".

Section 19.2 page 307.

Look up these terms in the glossary:

Arithmetic Operators
Assembly-Time Constant
Comment Field
Constant
Context Switch
CPU RAM
Destination Operand
Directives
Expressions
Immediate Instruction
Indexed Memory Addressing
Jump Instructions
Label Field
Location Counter
Mnemonic Code
Op-Code Field
Operand
Operand Field
Predefined Symbols
Pseudo-Instructions
Symbol
Symbol Table
Symbolic Addresses
Symbolic Memory Addressing
VDP RAM
Workspace Register Addressing

CHAPTER SIX
ASSEMBLING AND RUNNING A PROGRAM

Using The Editor/Assembler

The following is a brief overview of the steps you should take to enter, assemble, save, and run the sample program in Chapter Five. Read the documentation provided by TI with the Editor/Assembler product covering use of the Editor in preparing source programs. Take some time to become familiar with the editing capabilities which are available. Many of the features of the Editor program apply readily to the preparation of text and documents as well as assembly source statements. The steps given here assume that you have only one disk drive. If you have more than one, leave the diskette with the Editor/Assembler programs in drive #1 and save your source and object files on drive #2 or #3.

Get to the Editor/Assembler title screen, and with the software diskette in drive #1, select the EDIT option (#1). This should take you to the title screen for the Editor. From the Editor menu, select the EDIT option (#2). The Editor program will load from the diskette. Carefully type in the sample program using all upper-case characters. Check your typing. The sample program has been sucessfully assembled and run using both the Editor/Assembler and the Mini-Memory. If you encounter errors, it may be because you have entered something incorrectly.

When you have completed entering the source code, press ESCAPE (FCTN 9) twice to return to the EDITOR title screen. Then, insert the diskette on which you are going to save your source program into disk drive #1 and select the SAVE option (#3). The prompt VARIABLE 80 FORMAT (Y/N)? appears. "Y" indicates variable, "N" indicates fixed. The Editor/Assembler handles both variable or fixed length files For this exercise, answer "Y" to this prompt. Next you will be asked for the file name you want to call your source program. For this exercise, type in DSK1.SOURCE, and press ENTER. The contents of the text buffer (the SOURCE program you have just entered) will be saved on disk drive number one in variable 80 byte record format, as "DSK1.SOURCE". Next, remove your diskette and insert the software diskette which has the assembler program on it. Press ESCAPE (FCTN 9) to return to the Editor/Assembler title screen and select the ASSEMBLE option (#2).

You will be asked if you want to load the assembler at this point. This is done to allow you to check to be sure the proper diskette is in the drive. When you have the diskette in drive #1, reply "Y" to this prompt. The next prompt you should see will be for the name of the file which contains your source program. Remove the diskette containing the assembler and insert the diskette that you saved your source program on. Type in DSK1.SOURCE. The next prompt is for the name you want to give the object code that will be generated from your source code. For this example, type in DSK1.OBJECT. The next prompt is for a valid device name for the assembly listing. If you do not have a printer, you can save the listing on disk for later viewing with the editor program, or you can elect not to produce an assembly listing. If you have a printer, you should enter the parameters which describe the interface port you are using for your printer. (Examples: RS232.BA=1200, RS232/2.BA=9600.PA=N, PIO).

If you choose not to produce a printed listing or a list file, press enter at this prompt. The option to produce or not to produce a listing comes up in the next prompt. The next prompt asks for the options you want for this assembly. Option "R" allows you to refer to the general workspace registers in your program as R0,R1,R2.. R10, etc. Option "C" will produce object code in compressed format which takes up less storage than uncompressed code. Option "L" is the option to produce an assembly listing. If you do not choose option "L", no listing will be produced. Option "S" will include a symbol table map if you have also picked "L". For this exercise, the minimum option you will need is "R". Choose the other options depending on your own hardware configuration.

There is an additional option available that is not documented in the Editor/Assembler manual. This is the "T" option. This will print out the location and hexadecimal value of each byte of a TEXT string. Without the "T" option, the assembly listing will print only the beginning address (location counter value) of a TEXT string and the hexadecimal value of only the first byte of that string.

"Assembler Executing" should now appear as the assembler processes your code. If everything has gone right, the assembly process should end with 0000 ERRORS. If errors do occur, go back and re-examine the source code that you saved as DSK1.SOURCE. To edit your source code, place the software diskette into disk drive #1. Get to the Editor title screen and select the LOAD option (#1). After the editor program has loaded, there will be a prompt for file name. Insert the diskette containing your source program into disk drive #1 and type in DSK1.SOURCE. The editor will load your source program from the diskette. Select the EDIT option (#2) to review and edit your source program. When you are finished making corrections, save the source code under the same name, and repeat the assembly process.

Once you have successfully assembled the sample program, press Enter to return to the Editor/Assembler screen and select the LOAD AND RUN option (#3). The first prompt will ask you for the name of the file that contains your object program. Type in DSK1.OBJECT and the loader will load it from the diskette into memory. Another prompt for "File Name" will appear. This is because the loader provided with the Editor/Assembler will allow you to continue loading object programs until memory is full. There is only one program to load and run, so at this point press Enter. The next prompt will be for "Program Name." The name of this program and its entry point address was DEFined as "START". Type in START for this prompt and the program should run. If you coded END START as the last line of the program then the program will execute as soon as it is loaded without needing you to state the entry point address.

The numerals "43" should appear in the lower lefthand corner of the screen. To exit from the program's control, press QUIT (FCTN =). This should return you to the main title screen.

Remember, it is entirely possible to have made a mistake entering the sample program and still wind up with 0000 ERRORS at the end of the assembly process. When your assembly program runs, it is in control of the computer. It may be necessary to turn the computer off to stop a "runaway" program and then turn it back on.

Using the Line-by-Line Assembler

Follow the step by step instructions for loading the Line-by-Line assembler program from cassette. Get to the Mini-Memory title screen and select the RUN option (#2). The prompt PROGRAM NAME? appears. Type in NEW to get into the Line-by-Line assembler program.

The Line-by-Line assembler immediately assembles each assembly statement as you enter it and stores the resulting machine language instructions at the address indicated. Two columns of numbers appear to the left of each line displayed. The first column is the location counter. The values in this column are addresses. They are shown in hexadecimal notation. The Line-by-Line assembler starts out "pointing to " address >7D00.

The next column of numbers represents the value in hexadecimal notation stored at the address indicated. You are in effect, "peeking" into the computer and seeing the value presently stored at a specific address. The screen shows:

```
7D00 045B
```

Try pressing Enter several times and notice what happens. Each time you press Enter the location counter is incremented by two. The location counter advances two bytes, or one word, at a time. The hexadecimal value at each word of memory is displayed as well. The location counter always advances to an even address. Not only can you move forward like this, but you can also jump around to various locations too. To affect the location counter, use the assembler directive AORG. With AORG, you can "peek" at any location. Press the space bar and type:

AORG 7D00 and press ENTER.

The screen shows:

```
7D00 045B
```

So, you are back where you started. Now type:

AORG >7D0C and press Enter.

The screen shows:

```
7D0C A100
```

This will be the first entry in the sample program. In addition to peeking into the computer's memory, you can also alter the contents of any location with the Line-by-Line assembler. You can "poke" a number into any address. Type:

AORG >837C and press Enter.

The screen shows:

```
837C 2000
```

Now type in:

DATA >0000 and press Enter.

The screen now shows:

```
837C 0000 DATA >0000
837E 0000
```

The DATA directive you entered with a value of all zeroes initialized the address indicated by the location counter to all zeroes. The Line-by-Line assembler then advanced to the next even address, >837E. Address >837C is the address of the GPL status byte. When you first looked at the address of the GPL status byte, its value was >20. The very next byte's address is >837D which contained >00. The display showed 2000, which is the contents of the word made up of bytes >837C and >837D. When you entered the DATA >0000 directive you cleared (set to all zeroes) the status byte and the very next byte as well. Now type:

AORG >837C and press Enter.

The screen shows:

```
837C 2000
```

The status byte value is back to >20. One of the things the status byte can detect is the depressing of keys on the keyboard. Whenever any key is pressed, the value >20 will be present in the status byte. This knowledge will be applied in a later example. The DATA directive cleared the status byte, but as soon as you pressed the keys to type in "AORG", the status byte was reset by the computer to >20.

Now type AORG >7D0C. This is line 01 of the sample program. Enter the source code very carefully just as it appears in all upper case characters. If you make a mistake while entering a line and you realize it before you press Enter, type in "E" and press Enter. This informs the Line-by-Line assembler that you have made an error and allows you to re-enter the line immediately. You will naturally make some errors at first. Many errors will be detected by the Line-by-Line assembler program the moment you enter them much the same as TI BASIC syntax errors are detected. You may scroll backward and forward to view the code you have already entered by use of the up and down arrow keys. Refer to "Editing Techniques" in the Line-by-Line asssembler instruction book and take some time to learn to enter and correct source statements.

Lines 41 through 45 were inserted into this version of the sample program to add the program name and entry point to the REF/DEF table. At line 41, a calculation is needed to find out if there is sufficient room left to add the program name to the REF/DEF table. To do this, you need to find the difference between two values which are stored at addresses >701C and >701E. The value stored at address >701C is the First Free Address in the Module (FFAM). The value stored at address >701E is the Last Free Address in the Module (LFAM). Type:

```
  AORG >701C   and press Enter
```

The screen shows:

```
701C >7FB2
```

Now type in:

```
  DATA >7FB2   and press Enter
```

The screen shows:

```
701C >7FB2 DATA >7FB2
701E >7FE8
```

The effect of the DATA >7FB2 directive is to not change the value at >701C and advance the location counter to the next even word address, >701E, the other address to be checked.

```
       >7FE8
     - >7FB2
      ------
   = >0036
```

The answer to this calculation must be greater than 8 bytes. Otherwise, there is not enough space. The answer of >36 is easily large enough. The next step is to take the value at address >701E (in this case >7FE8), subtract 8 bytes from it and place the new value back at >701E.

```
   >7FE8
 - >0008
   -----
 = >7FE0
```

Since the location counter is currently pointing at address >701E, all that is needed is a DATA directive to place the new value there.
Type:

```
DATA >7FE0   and press Enter
```

The next entry will be AORG and the address you just calculated. Type in:

```
AORG >7FE0   and press Enter
```

This points to the REF/DEF table area into which will be placed the program name and entry point. The program name takes up 6 characters, which is entered as a string with the TEXT directive: Type in:

```
TEXT 'EXAMP1' and press Enter
```

If the program name used is less than six characters, the rest of the name entry in the REF/DEF table will be padded with spaces. The next required entry will be a DATA directive followed by the label which marks the entry point of the program. Entry point means the address of the first instruction to be executed in the program. Reading an assembly program from top to bottom, the entry point may, or may not, be the first program instruction in the program sequence. In the sample program the label for the entry point address was "ST". Type in:

```
DATA ST    and press Enter
```

The label "ST" will be equated with the entry point address in the program named EXAMP1.

```
END
```

The effect of the END directive with the Line-by-Line assembler is to signal an end to the source code and the assembly process. The moment you type END, you will exit from the assembler. The use of "END START" is not possible for programs entered using the Line-by-Line assembler. When you type END and then press Enter, there should not be any unresolved references. Before you type in END, scroll back over the source program you have just entered by use of the up and down arrow keys. If you have entered the sample program exactly as shown, there should not be any problems.

Once you have successfully assembled the sample program, go to the Mini-Memory title screen and select the RUN option (#2). The prompt "Program Name?" appears. Type in EXAMP1. The run program will look in the REF/DEF table, find the name EXAMP1, and branch to the entry point of the program (symbolic address ST) also found in the REF/DEF table.

The numerals "43" should appear in the lower lefthand corner of the screen. To exit from the program's control, press QUIT (FCTN =). This should return you to the main title screen.

Remember, it is entirely possible to have made a mistake entering the sample program and still wind up with 0000 unresolved references at the end of the assembly process. To repeat, when you run your assembly program, it is in complete control of the computer. It may be mecessary to turn your computer off to stop a "runaway" program, and then turn it back on.

EDITOR/ASSEMBLER MANUAL REFERENCES

The following references will provide you with some more information on assembling and running programs.

Section 1.1 page 15 through Section 2.5 page 38
Section 15.1 page 235 through Section 15.1.1 page 236
Section 15.5 page 243 through Section 15.5.2 page 245
Section 19.1 page 305 through Section 19.2 page 307

Look up these terms in the glossary:

Assembler
Assembly Options
Command Mode
Edit Mode
Editor
End-of-File Marker
Fatal Error
List File
Loader
Loading
Non-Fatal Error
Special Keys
Symbol
Symbol Table
Window

CHAPTER SEVEN
SCREEN AND CHARACTER DISPLAYS

One of the things you can do right away with a language such as TI BASIC is to display numbers, letters, and other symbols on the screen and control their position and color. The sample program introduced two fundamental Video Display Processor routines. The numerals "43" were displayed at the lower left corner of the screen as its only output. The numerals appeared as black symbols on a green background. These are the preset colors of the screen and displayable character sets which are in effect when your assembly is run. This chapter will examine the TMS9900 assembly language approach to screen displays more closely.

The processing necessary to generate the video signals that create symbols and graphics is handled by a separate microprocessor known as the TMS9918 chip in the older 99/4 and as the TMS9918A in the 99/4A computers. The central processor in your computer is the TMS9900 chip. It is the microprocessor dealt with directly in TMS9900 assembly language. Actually, your Texas Instruments home computer contains several distinct microprocessors which work in concert to make the home computer a reality.

The area within memory which is designated for the Video Display Processor's needs is a specially segregated area known as VDP Random Access Memory (RAM). TMS9900 assembly instructions like MOVe, or Add, will not work on addreses within VDP RAM. Data must be manipulated within your program's domain (CPU RAM) and written to or read from VDP RAM by means of special routines which allow these two processors to share data and communicate with each other. The first program example introduced two very handy VDP routines, VSBW (VDP Single Byte Write), and VMBW (VDP Multiple Byte Write), which copied data from your program into VDP RAM. The data was first set up within the program. Then, by placing certain required values into the proper registers, and branching to the routine's address, the screen was cleared, and then the sum of 10 plus 33 was displayed. This pattern of loading certain key registers with parameter values and branching to a special address is a recurring one throughout the various VDP routines. To use any VDP routine in your program you must include a REF directive (Editor/Assembler) or an EQUate directive (Line-by-Line assembler) with the symbolic name for each VDP routine you want to use.

One of the easiest ways to display titles and messages combines the use of the TEXT directive and the Multiple Byte Write routine. With the Editor/Assembler, this routine's symbolic address is equated to VMBW. The Line-By-Line program example equated the name as VM. With the TEXT directive memory can be initialized in easily readable character strings. The way the string appears in the program is exactly how it will look when properly displayed. Here is a program segment to accomplish this:

```
MSG1 TEXT '** PROGRAM NUMBER 2 **'
```

The character string is enclosed by single quote (') marks. The single quote marks will not be part of the display. Any characters can be used with the TEXT directive except the single quote mark. The single quote mark can only be used to delimit the contents of the string. To display this program title you must give VMBW three pieces of information:

1. The address in VDP RAM you want this message to be placed at (written to). The position on the screen.

2. The begining address of the message. Where to find the message.

3. The number of bytes to write. The length of the message.

In TI BASIC, the screen can be addressed as row N, column N. There are 24 rows and 32 columns. This same screen configuration is defined in assembly language as a table of 768 bytes. This is the Screen Image Table In VDP RAM. Each byte of this table represents one screen position. The VDP RAM addresses for these 768 bytes are 0 through 767 decimal and >00 through >2FF hexadecimal. Use this formula to determine the proper VDP RAM address for a given set of row and column values:

VDP RAM ADDRESS DECIMAL = ((ROW - 1) * 32) * (COLUMN - 1)

To display the above message at row 10, column 6:

```
ADDRESS = ( ( 10 - 1) * 32 ) + ( 6 - 1)
        =   (  9  *  32  )   +    5
        =        288         +    5
-----------------------------------------
        =          293
```

To pass this information to VMBW, place it into register 0 with a LI (Load Immediate) instruction:

```
LI R0,293
```

The next piece of information required by VMBW is the beginning address of the message. The label "MSG1" was included with the TEXT directive and is therefore equated to the value of the beginning address of the message. Remember that this value is the ADDRESS of the message, not the data itself. The VMBW routine needs this information in register 1. Again using the LI instruction:

```
LI R1,MSG1
```

Finally, VMBW needs to know how long the message is in bytes. One byte is required for each character of the message. There are 22 characters in the message labeled MSG1. This value must be put into register 2:

```
LI R2,22
```

Since the location counter always advances to an even word address, it is always a good idea to set aside storage with directives like TEXT in even byte amounts.

To display the message, perform a Branch and Load Workspace Pointer to the address of the VMBW routine:

```
BLWP @VMBW
```

The program segments:

```
       REF   VMBW
       •
       •
MSG1   TEXT '** PROGRAM NUMBER 2 **'
       •
       •
DISP   LI    RØ,293
       LI    R1,MSG1
       LI    R2,22
       BLWP @VMBW
```

Of course the TEXT directive is just one way to build displayable data and must be "hard coded" into the program. You can also use the ASCII codes for letters and numbers and have the program create numeric and alphanumeric strings just as the sample program did with the answer to its addition problem:

```
PNTANS  BSS 2
•
•
        MOV  R5,@PNTANS
•
•
        MOVB R5,@PNTANS
•
•
PUTUP   LI   RØ,738
        LI   R1,PNTANS
        LI   R2,2
        BLWP @VMBW
```

The VSBW (Single Byte Write) routine only writes one byte at a time. Since the length of the data to be written is always 1, R2 is not needed for VSBW. You need only give VSBW the correct VDP address in R0, and the data to be written in the first 8 bits (left byte) of register 1. This is different from VMBW where register 1 must contain the address of the data.

Suppose that instead of the title MSG1, You want to display an asterisk (*) at row 10, column 6. The ASCII code for an asterisk is 42 decimal, >2A hexadecimal. You have already calculated the proper address in VDP RAM for row 10, column 6, as 293. The code would look like this:

```
      LI     RØ,293
      LI     R1,>2AØØ
      BLWP   @VSBW
```

A TI BASIC statement you should be familiar with is CALL SCREEN (n). The value of n is some number between +1 and +16. Each number represents a different screen color. In TMS9900 assembly language the same set of colors are available to you. Their values are all one less than their TI BASIC counterparts or 0 through 15 decimal, >0 through >F hexadecimal. The color of the screen border is controlled by VDP RAM Write Only Register 7. Access to this register and control of screen border color is accomplished through the VDP Write To Register subroutine equated by the Editor/Assembler to VWTR. With the Line-by-Line assembler this routine's address of >6034 must be EQUated to some 2 character label. The colors and their respective TMS9900 assembly language hexadecimal values are:

TRANSPARENT	>Ø	MED. RED	>8
BLACK	>1	LIGHT RED	>9
MED. GREEN	>2	DARK YELLOW	>A
LIGHT GREEN	>3	LIGHT YELLOW	>B
DARK BLUE	>4	DARK GREEN	>C
LIGHT BLUE	>5	MAGENTA	>D
DARK RED	>6	GRAY	>E
CYAN	>7	WHITE	>F

To set the border color of the screen to magenta, the TMS9900 assembly instructions would be:

```
      LI     RØ,>Ø7ØD
      BLWP   @VWTR
```

Register 0 contains all the information that VWTR needs. The left byte (reading from left to right, the first and second hex digits >07 of register 0 tells VWTR which VDP register to write to. In this example, it is VDP register 7. The VDP registers are single byte registers (8 bits) unlike the general workspace registers which are full word (16 bit) registers. The right byte (hex digits >0D) contains the value of the color you want. Of these 8 bits, the least significant 4 (hex digit >D) set the screen border color. The most significant 4 bits (hex digit >0) set the foreground color when the TMS9918A is in text mode. Text mode is another form of display available with the TMS9918A. When you reach a proficient level with TMS9900 assembly language and VDP handling, you may want to try other modes. For right now though, the value you place in this position is of no consequence. The display mode your home computer operates in while in BASIC, Extended BASIC, and most applications is the graphics mode. Learn to master this display mode before you attempt to use any others.

The effect of setting VDP RAM register 7 to >0D is to generate bands of magenta color at the top and bottom of the screen. The actual effect of the TI BASIC CALL SCREEN(n) instruction involves not only setting the color of the border, but setting the background colors of any characters that might be displayed to the same color. In this way, no matter what is displayed on the screen, there will be one uniform background color for the entire display. The foreground and background colors of the TI Home Computer character set are controlled by an area in VDP RAM known as the color table.

Each entry in the color table is made up of one byte of data. Each byte controls the foreground and background colors of a set of 8 characters. The color table is a relocatable table. That is, with certain TMS9900 instructions, it is possible to change the location within VDP RAM that the table will occupy. Changing the color table's location is only necessary for other display modes. For right now, do not relocate the color table. Instead, use its default VDP RAM beginning address of >0380.

To change the foreground and background colors of a particular character, the corresponding address within the color table must be determined, and one byte of data must be placed into that address. To emulate CALL SCREEN(n), it is necessary to change the background color while leaving the foreground color at its default value of >1 (black). The value for black on magenta would be >1D. The left 4 bits of the byte control the foreground color (>1, black), and the right 4 bits control the background color (>D, magenta).

Of course this data must be placed into the correct color table address to yield the desired result. Here is a handy chart which details the color table. The addresses given are displacement values. Each value from the chart must be added to the beginning VDP RAM address of the color table. In the case of the example, operating in graphics mode and not having done anything to relocate the color table, the beginning VDP RAM address of the color table is >0380.

COLOR TABLE REFERENCE CHART

COLOR TABLE DISPLACEMENT	CHARACTER CODES AFFECTED		
>00	>00	THROUGH	>07
>01	>08		>0F
>02	>10		>17
>03	>18		>1F
>04	>20		>27
>05	>28		>2F
>06	>30		>37
>07	>38		>3F
>08	>40		>47
>09	>48		>4F
>0A	>50		>57
>0B	>58		>5F
>0C	>60		>67
>0D	>68		>6F
>0E	>70		>77
>0F	>78		>7F
>10	>80		>87
>11	>88		>8F
>12	>90		>97
>13	>98		>9F
>14	>A0		>A7
>15	>A8		>AF
>16	>B0		>B7

>17	>B8	>BF
>18	>CØ	>C7
>19	>C8	>CF
>1A	>DØ	>D7
>1B	>D8	>DF
>1C	>EØ	>E7
>1D	>E8	>EF
>1E	>FØ	>F7
>1F	>F8	>FF

The character codes for the entire range of possible characters start at >00 and end at >FF. Referring to the chart above, the color table address displacement value for character >FF = >1F. Add each of the displacement values to the beginning address of >0380:

```
  >Ø38Ø            >Ø38Ø

+ >ØØØØ          + >ØØ1F
--------         --------

= >Ø38Ø          = >Ø39F
```

This demonstrates that to affect the colors of this range of characters, one byte of color data must be placed at VDP RAM addresses >0380 through >039F. Since one byte of data is required at each address, the Single Byte Write routine will be used. The instructions to do this are:

```
       LI    RØ,>Ø38Ø  Load RØ with first address in VDP RAM
       LI    R1,>1DØØ  Place color codes into left byte of R1
PUTCOL BLWP  @VSBW     Write left byte of R1 to address in RØ
       INC   RØ        Add 1 to the address in RØ
       CI    RØ,>Ø39F  See if it has gone too far
       JLE   PUTCOL    If not, repeat the process
```

The last instruction used was the Jump if Low or Equal. The previous Compare Immediate is checking R0 for the last address value to be written to. The JLE instruction completes the comparison by directing program logic to keep returning to the loop PUTCOL as long as the address value in R0 is less than or equal to >039F.

The steps outlined so far to affect the color table entries are designed to mimic the effect of the TI BASIC CALL SCREEN(n) command. Another TI BASIC command is CALL COLOR(s,f,b). The "s" represents the set of characters to be affected, "f" the foreground color, and "b" the background color. Set may be a number from +1 to +14. In TI BASIC, these color sets are equivalent to color table displacement values >04 through >11. To specify a color combination of white on dark blue for the asterisk (character code 42 decimal, >2A hex) the TI BASIC statement CALL COLOR(2,16,5) would be used. The asterisk is a member of character set 2 in TI BASIC and the TI BASIC color codes for white and dark blue are 16 and 5 respectively.

To accomplish the same result in TMS9900 assembly language, first consult the color table reference chart above and find the range of character code values to which the asterisk belongs. >2A falls within the range of values >28 through >2F. The displacement value for this set is >05. Next, add this displacement value to the beginning address of the color table to determine the correct address (>0380 + >05 = >0385).

The color values in TMS9900 assembly language are all one less than in TI BASIC. White is 15 decimal, >F hexadecimal. Dark blue is 4 decimal, >4 hexadecimal. The instructions to set the color of the asterisk to white on dark blue would be:

```
LI   R0,>0385  COLOR TABLE ADDRESS OF ASTERISK
LI   R1,>F400  COLOR VALUES, WHITE (F) ON BLUE (4)
BLWP @VSBW     WRITE ONE BYTE OF DATA
```

No program loop is involved because this example only affects the foreground/background colors of one set of characters. Notice from the color table reference chart that there are color table entries for character codes which are less than and greater than the range of ASCII characters (30 through 126 decimal, >1E through 7E hexadecimal). These character values are not defined as displayable characters. Some of these character codes represent ASCII control characters which are used to regulate communications between computers. Others have no definition at all. Through TMS9900 assembly language you can make use all these character values in a variety of ways.

First, recall the example program and its CLEAR routine. The standard method of clearing the screen display is to fill the entire screen with spaces (ASCII code 32 decimal, >20 hexadecimal). But the CLEAR routine in the sample program wrote the >00 character code to the screen. Since character code >00 is not defined as a displayable symbol, the effect looks the same as using the space character code.

Another way to use these extra character codes is for color graphics. Recall from TI BASIC that if the foreground/background colors of a character are set to the same color, anytime that character is displayed, a solid block of color would appear. One way to apply this would be the creation of a border of color around the screen. Each row of the screen has 32 columns. To create the side borders, use columns 1,2,31,32 of each row. Set the characters used to fill these positions to the same color as the top and bottom screen borders. So that you can still be able to display and use the standard ASCII character set, use a character code which is outside the ASCII range and will not be needed for anything else.

Here are the TMS9900 assembly language instructions. Note that the DATA directive can be use to intialize more than one word at a time.

```
BORDER  DATA >8080,>2020,>2020,>2020
        DATA >2020,>2020,>2020,>2020
        DATA >2020,>2020,>2020,>2020
        DATA >2020,>2020,>2020,>8080
*Define 32 characters to be used to fill
*each row on screen. Characters occupying
*columns 1,2,31,32 are greater than any
*ASCII codes and characters in remaining
*columns are ASCII spaces.
```

```
          LI   RØ,>Ø7Ø6    Set top and bottom screen borders
          BLWP @VWTR        to dark red
          LI   RØ,>Ø39Ø    Set the color of character >8Ø to
          LI   R1,>66ØØ     dark red on dark red
          BLWP @VSBW
          LI   RØ,>Ø383    Set range of ASCII characters to
          LI   R1,>1FØØ    black on white. Space character (>2Ø)
CLOOP     BLWP @VSBW       will appear white.
          CI   RØ,>Ø38F
          JEQ  BPUT
          INC  RØ
          JMP  CLOOP
BPUT      LI   RØ,Ø        Fill screen with BORDER pattern.
          LI   R1,BORDER
          LI   R2,32
BLOOP     BLWP @VMBW
          CI   RØ,736
          JEQ  EXIT
          AI   RØ,32       Add Immediate adds 32 to RØ to
          JMP  BLOOP       address the next row.
EXIT      .........

    ( rest of program )
```

Here is the effect on the screen:

```
 ____________________________
|                            |
|            RED             |
|  ------------------------  |
|  |                      |  |
|  |                      |  |
|  |                      |  |
|R |                      |R |
|E |                      |E |
|D |        WHITE         |D |
|  |                      |  |
|  |                      |  |
|  |                      |  |
|  ------------------------  |
|            RED             |
 ----------------------------
```

The patterns used to generate characters are controlled by another table in VDP RAM called the pattern descriptor table. The pattern descriptor table is also a relocatable table. The default beginning address in VDP RAM for the pattern descriptor table is >0800. Each table entry takes up 8 bytes. By changing the values stored in the pattern descriptor table, you can create graphics and symbols of your own. You can redefine the ASCII character set or use any of the other available characters. For a complete explanation of creating patterns see the TI BASIC or Extended BASIC reference material on the CALL CHAR subprogram. Access to the pattern descriptor table is much like that of the color table. To redefine a character or create a graphic, the data which describes the pattern must be placed into the corresponding address in the pattern descriptor table for the character code used. To help get you started, here is a partial list of pattern descriptor table displacement values and their respective character codes. With a little arithmetic, you should be able to determine the address you want.

PATTERN DESCRIPTOR TABLE REFERENCE CHART

(ALL VALUES ARE HEXADECIMAL)

PATTERN DESCRIPTOR TABLE DISPLACEMENT	CHARACTER CODE AFFECTED
>ØØØ	>ØØ
>ØØ8	>Ø1
•	•
>ØFØ	>1E
•	•
>1ØØ	>2Ø
>1Ø8	>21
>11Ø	>22
•	•
>15Ø	>2A
>158	>2B
>16Ø	>2C
>168	>2D

If you multiply the value of a charcter code by 8, you will find the displacement value in the pattern descriptor table for that character. This value must be added to the beginning address in VDP RAM for the pattern descriptor table. Unless you relocate the table by means of special instructions, the table starts at >0800 hexadecimal, 2048 decimal. To create a new symbol for the cursor, for example, you would first multiply the the character code for the cursor by 8:

```
                    HEX            DECIMAL

CURSOR =            >1E               3Ø

                  x  >8            x   8
                  ------           -----
                  = >FØ            = 24Ø
```

Then, add this answer to the beginning VDP RAM address:

```
Beginning PDT address   >Ø8ØØ         2Ø48

Displacement value    + >ØØFØ       +  24Ø
                      --------      -------
Address desired       = >Ø8FØ       = 2288
```

You now know that the address of the pattern descriptor table entry for the pattern of the cursor character is >08F0. The TMS9900 assembly language instructions to change the cursor pattern are:

```
CURPAT  DATA  >ØØ7E,>4242,>4242,>7EØØ

                          *DEFINE 8 NEW BYTES OF DATA
                          *TO DESCRIBE NEW PATTERN
        LI    RØ,>Ø8FØ    *LOAD VDP RAM ADDRESS INTO RØ
        LI    R1,CURPAT   *LOAD ADDRESS OF DATA INTO R1
        LI    R2,8        *LOAD DATA LENGTH INTO R2
        BLWP  @VMBW       *WRITE DATA TO PDT
```

Now, anytime the cursor symbol is displayed with an instruction set such as these:

```
        LI    RØ,293
        LI    R1,>1EØØ
        BLWP  @VSBW
```

The pattern that was defined will be displayed instead of the standard cursor symbol. Note that the character code for the cursor of >1E is still used. The computer takes the character code you specified, looks in the color table for the correct foreground/background colors, looks in the pattern descriptor table for the pattern to be displayed, and displays that pattern/color combination at the specified screen address.

Screen and character displays created through TMS9900 assembly language programs are not difficult once you master the fundamentals outlined in this chapter. The amazing speed of the TMS9900 assembly language becomes evident when it is used with Video Display Processor applications. Changes to the screen display happen almost instantaneously. Many more graphics capabilities become available through TMS9900 assembly language than are even possible in TI BASIC. Here is a complete TMS9900 assembly language program which will demonstrate some of the principles covered in this chapter:

```
        DEF  START
        REF  VWTR,VSBW,VMBW
WR      BSS  >20
RETURN  BSS  2
STATUS  EQU  >837C
BORDER  DATA >8080,>2020,>2020,>2020
        DATA >2020,>2020,>2020,>2020
        DATA >2020,>2020,>2020,>2020
        DATA >2020,>2020,>2020,>8080
MSG1    TEXT '** PROGRAM NUMBER 2 **'
START   MOV  R11,@RETURN SAVE RETURN ADDRESS
        LWPI WR          LOAD WORKSPACE POINTER
        LI   R0,>0706    SET BORDER COLOR TO
        BLWP @VWTR         DARK RED
        LI   R0,>0390    SET COLOR OF >80 CHARACTER
        LI   R1,>6600      TO DARK RED
        BLWP @VSBW           ON DARK RED
        LI   R0,>383     SET RANGE OF ASCII DISPLAY
        LI   R1,>1F00      CHARACTERS COLOR TO
CLOOP   BLWP @VSBW          BLACK ON
        CI   R0,>038F         WHITE
        JEQ  BPUT
        INC  R0
        JMP  CLOOP
BPUT    LI   R0,0        LOAD THE SCREEN IMAGE
        LI   R1,BORDER     TABLE WITH THE
        LI   R2,32           BORDER PATTERN
BLOOP   BLWP @VMBW
        CI   R0,736
        JEQ  EXIT
        AI   R0,32
        JMP  BLOOP
EXIT    LI   R0,293      DISPLAY THE PROGRAM
        LI   R1,MSG1       TITLE
        LI   R2,22
        BLWP @VMBW
        CLR  @STATUS     CLEAR THE GPL STATUS BYTE
        MOV  @RETURN,R11 GET RETURN ADDRESS
        DECT R11         ALTER THE RETURN ADDRESS
        RT               RETURN
        END
```

Follow the instructions in Chapter Six for assembling and running this program. Since this program alters the return address in the same way that the first sample program did, it, too, will "freeze up" in order to allow you to view the results. Press QUIT (FCTN =) to exit from program control. The sample program is written for the Editor/Assembler. To code and run this program using the Line-by-Line assembler, refer to Chapter Five for the steps to make the changes required. Here is the listing for the Line-by-Line assembler:

```
* SCREEN DISPLAY PROGRAM EXAMPLE *
* MINI-MEMORY VERSION            *
*                                *
        AORG >7D0C
VW      EQU  >6034
VS      EQU  >6024
VM      EQU  >6028
WR      BSS  >20
BD      DATA >8080,>2020,>2020,>2020
        DATA >2020,>2020,>2020,>2020
        DATA >2020,>2020,>2020,>2020
        DATA >2020,>2020,>2020,>8080
M1      TEXT '** PROGRAM NUMBER 2 **'
GO      MOV  R11,R10
        LWPI WR
        LI   R0,>0706
        BLWP @VW
        LI   R0,>0390
        LI   R1,>6600
        BLWP @VS
        LI   R0,>383
        LI   R1,>1F00
CL      BLWP @VS
        CI   R0,>038F
        JEQ  BP
        INC  R0
        JMP  CL
BP      LI   R0,0
        LI   R1,BD
        LI   R2,32
BL      BLWP @VM
        CI   R0,736
        JEQ  EX
        AI   R0,32
        JMP  BL
EX      LI   R0,293
        LI   R1,M1
        LI   R2,22
        BLWP @VM
        CLR  @>837C
        MOV  R10,R11
        DECT R11
        RT
        END
```

EDITOR/ASSEMBLER MANUAL REFERENCES

The following references will provide you with some more information on screen and character displays.

Section 16.1 page 246 through Section 16.1 page 248
Section 21.1 page 325 through Section 21.2.3 page 330
Section 21.7 page 342 through Section 21.7.1 page 342
Section 24.7 page 428

Look up these terms in the glossary:

Character Constant
Character Set
Character String
Color Table
Undisplayable Characters
Utilities
VDP RAM

CHAPTER EIGHT
PROCESSING KEYBOARD INPUT

Accepting and processing data entered by the computer user from the keyboard is always an important program function. The acceptance of keyboard input also implies manipulating the screen display as well. Whenever you enter data by pressing keys you expect to see the characters you are typing displayed upon the screen as they are entered. In addition to entering data, you are accustomed to using special function key combinations to control operation of the computer.

Several methods of accomplishing this were available to you in TI BASIC. INPUT X is a TI BASIC statement which can be used to input a numeric value. CALL KEY(X,Y,Z) is another statement that can detect specific key strokes. These simple TI BASIC commands are capable of performing tasks which are far more complex than is apparent from the TI BASIC syntax that invokes them.

The TMS9900 assembly approach to these objectives involves reading from and writing to VDP RAM and the use of a special routine called the Keyboard Scan Utility. This routine is accessed by including a REF KSCAN directive when programming with the Editor/Assembler or by EQUating the routine's address of >6020 to some 2 character label when using the Line-By-Line assembler. As with most of the preceeding subroutines dicussed, KSCAN needs to use the BLWP instruction to activate the utility.

Besides the routine itself, there are also some special addresses you need to know about to make effective use of KSCAN. The value of the byte at address >8374 controls which keyboard device is to be scanned. A value of >00 scans the entire keyboard. A value of >01 scans the left side of the keyboard including joystick number one. Values from the joystick are placed at addresses >8376 (Y value) and >8377 (X value). A value of >02 at address >8374 scans the right side of the keyboard and joystick number two. The values from joystick number two are placed at the same addresses as number one (>8376,>8377). The normal or default value at >8374 is >00 (scan entire keyboard). Another address is >837C, which has been used before, and is the address of the GPL status byte. Whenever a key is pressed that is different from the key pressed the last time KSCAN was called, bit 2 of the GPL status byte is turned on. The value of the key pressed is placed at address >8375. If no key was pressed, address >8375 contains >FF.

When data is keyed into the computer, you expect to see a cursor on the screen marking the start of the input field. As the data is entered, you expect the cursor to move to the right and the data itself to be displayed where the cursor was. Lastly, if a mistake is made entering the data, you would like to be able to back up and re-enter data as long as Enter has not been pressed.

Here is a fundamental subroutine for accomplishing the above scenario. Its name is CURSOR. It assumes that EQUates have been included for KEYADR EQU >8374, KEYVAL EQU >8375, STATUS EQU >837C and that BYTE directives are included for ENTERV BYTE >0D, LEFTV BYTE >08, RITEV BYTE >09, ANYKEY BYTE >20. For Line-By-Line assembler, use DATA directives such as EV DATA >0D00, LV DATA >0800, etc.

```
*******************************************************
* "CURSOR" SUBROUTINE. INPUT: R0 = SCREEN ADDRESS OF  *
*  RESPONSE, R10 = MAX LENGTH OF RESPONSE. OUTPUT: R9 *
*  = LAST KEYSTROKE VALUE, R7 = ACTUAL LENGTH OF      *
*  RESPONSE, RESPONSE DATA BEGINS AT R0 FOR A LENGTH  *
*  OF R7                                              *
*******************************************************
CURSOR  CLR   R9
        MOV   R10,R10
        JEQ   SCAN
        CLR   @KEYADR
        LI    R1,>1E00
        BLWP  @VSBW
        MOV   R0,R8
        A     R8,R10
        MOV   R8,R7
SCAN    CLR   @STATUS
        BLWP  @KSCAN
        CB    @ANYKEY,@STATUS
        JNE   SCAN
        MOV   R10,R10
        JNE   ENTCHK
        RT
ENTCHK  CB    @ENTERV,@KEYVAL
        JEQ   ENTER
        CB    @LEFTV,@KEYVAL
        JEQ   LEFT
        CB    @RITEV,@KEYVAL
        JEQ   RITE
        C     R7,R10
        JEQ   SCAN
        MOV   R7,R0
        MOVB  @KEYVAL,R1
        MOVB  @KEYVAL,R9
        BLWP  @VSBW
        INC   R7
CURPUT  MOV   R7,R0
        LI    R1,>1E00
        BLWP  @VSBW
        B     @SCAN
LEFT    C     R7,R8
        JEQ   SCAN
        MOV   R7,R0
        LI    R1,>2000
        BLWP  @VSBW
        DEC   R7
        JMP   CURPUT
RITE    C     R7,R10
        JEQ   SCAN
        MOV   R7,R0
        LI    R1,>2000
```

```
45          BLWP  @VSBW
46          INC   R7
47          JMP   CURPUT
48 ENTER    LI    R1,>2ØØØ
49          MOV   R7,RØ
5Ø          BLWP  @VSBW
51          S     R8,R7
52          RT
```

Before dissecting this subroutine, here is how you would use it in your program. Before performing a BL @CURSOR, place into R0 the beginning screen address where you want the keyed input to appear. Into R10 place the maximum length of the data to be accepted. If you want to emulate a "PRESS ANY KEY" situation, place a length value of 0 into R10. The CURSOR routine will return to your program as soon as any key is pressed without capturing any data at all. Quite often when you want the computer user to respond to a prompt, the answer to be given is a one digit value. An example of this would be when the user is to respond "Y" or "N" or pick a number or letter from a menu. CURSOR always places the value of the last key pressed before Enter was pressed into the left byte of R9. For one digit replies, the reply value is available in R9 after returning from CURSOR without any moves or other manipulations. R7 will contain the actual length of the data which was entered. The actual number of characters keyed in may or may not be the same as the maximum allowable.

```
Ø1 CURSOR   CLR   R9
Ø2          MOV   R1Ø,R1Ø
Ø3          JEQ   SCAN
Ø4          CLR   @KEYADR
Ø5          LI    R1,>1EØØ
Ø6          BLWP  @VSBW
```

Line 1 of CURSOR clears register 9. Lines 2 and 3 check R10 for a value of zero. If a register, word, or byte is moved to itself, and the value of the item is zero, then the equal bit is set in the status register. Line 3 jumps to the label SCAN if R10 is zero. Line 4 clears addresses >8374 & >8375. The CLR instruction clears (sets to all zeroes) a full word of memory. Symbolic addresses KEYADR was EQUated to >8374. The CLR instruction clears this byte and the very next byte (>8375) also. >8375 is the address of KEYVAL. With one instruction, you have specified that you want to scan the entire keyboard, and clear any previous key stroke value. Line 5 loads R1 with the character code for the cursor symbol (>1E). Line 6 writes the cursor symbol to the screen address you specified in R0 before you branched to CURSOR.

```
Ø7          MOV   RØ,R8
Ø8          A     R8,R1Ø
Ø9          MOV   R8,R7
1Ø SCAN     CLR   @STATUS
```

Line 7 saves the beginning cursor address in R8. Line 8 adds the beginning cursor address in R8 to the field length value in R10 to determine the maximum cursor address. Line 9 moves the beginning cursor address into R7, which will be used as an accumulator of cursor address values. Line 10 clears the GPL status byte. You need to start out with all zeroes in the status byte in order to detect any keystroke.

```
11          BLWP  @KSCAN
12          CB    @ANYKEY,@STATUS
13          JNE   SCAN
14          MOV   R1Ø,R1Ø
15          JNE   ENTCHK
16          RT
```

Line 11 invokes the Keyboard Scan Utility. Line 12 uses the Compare Bytes (CB) instruction to compare the value of the GPL status byte to >20. This is the value that will be present in the GPL status byte if any key has been pressed. At line 13 the Jump if Not Equal instruction completes the comparison by returning to the label SCAN if no key has been pressed. Line 14 checks R10 for zero again by moving it to itself, and if R10 is not equal to zero, it then proceeds to the label ENTCHK. If R10 is equal to zero, line 16 returns (B * R11) to the calling program.

```
17 ENTCHK   CB    @ENTERV,@KEYVAL
18          JEQ   ENTER
19          CB    @LEFTV,@KEYVAL
2Ø          JEQ   LEFT
21          CB    @RITEV,@KEYVAL
22          JEQ   RITE
23          C     R7,R1Ø
24          JEQ   SCAN
25          MOV   R7,RØ
26          MOVB  @KEYVAL,R1
27          MOVB  @KEYVAL,R9
28          BLWP  @VSBW
29          INC   R7
```

CURSOR jumps to line 17 if some key was pressed and R10 is not equal to zero. The first order of business is to determine if any special keys have been pressed. If the user has pressed Enter, they have finished entering data. The value at >8375 when the Enter key is pressed is >0D. If the user wishes to correct his typing, he may press either the left or right arrow keys (FCTN S, FCTN D). The key value of the left arrow is >08, and the key value of the right arrow is >09. Lines 17 through 22 check for these conditions. Line 23 uses the Compare Words instruction to compare the cursor address accumulator (R7) to the maximum cursor address (R10). If they are equal, then the maximum allowable length of the data has already been reached. When this is true, no more data is accepted by CURSOR, and the only key stroke values CURSOR will accept are Enter or left arrow. If the maximum has not been reached, lines 25,26, and 27 accept the key data. Line 25 copies the address at which the data will be displayed from R7 into R0. Line 26 moves the value of the key stroke into the left byte of R1. Line 27 saves the key stroke value in R9. Line 28 writes the character code (key stroke value) to the screen. Line 29 increments R7, which is the new address of the cursor symbol.

```
3Ø CURPUT  MOV    R7,RØ
31         LI     R1,>1EØØ
32         BLWP   @VSBW
33         B      @SCAN
```

Line 30 places the new screen address that the cursor will occupy into R0. Lines 31 and 32 write the cursor symbol to the screen. The visual effect is that the cursor has moved one space to the right, and the character keyed in appears at the previous position of the cursor symbol. Line 33 branches to the label SCAN to repeat the entire process and form the loop.

```
34 LEFT    C      R7,R8
35         JEQ    SCAN
36         MOV    R7,RØ
37         LI     R1,>2ØØØ
38         BLWP   @VSBW
39         DEC    R7
4Ø         JMP    CURPUT
41 RITE    C      R7,R1Ø
42         JEQ    SCAN
43         MOV    R7,RØ
44         LI     R1,>2ØØØ
45         BLWP   @VSBW
46         INC    R7
47         JMP    CURPUT
48 ENTER   LI     R1,>2ØØØ
49         MOV    R7,RØ
5Ø         BLWP   @VSBW
51         S      R8,R7
52         RT
```

Lines 34 through 52 detail the actions to be taken when one of the special keys is pressed. LEFT moves the cursor to the left and fills the field with blanks. Line 34 checks to see if the current cursor address (R7) is already at the minimum value (beginning cursor address). If the cursor is as far left as it can go, no action can be taken. RITE does just the opposite of LEFT. ENTER is the label the program jumps to when it is determined that the user has pressed Enter, signaling an end to the input of data. Lines 48,49, and 50 remove the cursor from its last known screen position. Line 51 uses the Subtract words instruction. The contents of R8 are subtracted from the contents of R7, and the answer is placed in R7. This action subtracts the beginning cursor address (R8) from the last cursor address (R7). The difference between the two is the actual length of the data which was keyed in. The RT instruction at line 52 returns to the calling program.

Suppose that a particular application requires that the computer user enter their full name. The maximum length of data that will be accepted has been determined to be 30 letters (30 bytes). It has also been determined that the data is to be accepted at row 10, column 1. Here are the program segments which prompt and accept this data.

```
PROMPT  TEXT  'ENTER FULL NAME '
        .
        .
        LI    R0,256     PUT UP PROMPT MESSAGE AT
        LI    R1,PROMPT  ROW 9, COLUMN 1
        LI    R2,16
        BLWP  @VMBW
        LI    R0,288     ROW 10, COLUMN 1
        LI    R10,30     LENGTH OF DATA
        BL    @CURSOR    GET THE DATA
```

At this point, the name entered is displayed on the screen beginning at row 10, column 1, and resides in VDP RAM at addresses 288 through 317, providing that the actual data length is 30. To make use of this data in the program, you need to get it from VDP RAM into the program. To accomplish this, you will need the VDP Multiple Byte Read routine (VMBR). Alternatively, you might also want to use the VDP Single Byte Read (VSBR). These routines operate much the same as VMBW and VSBW do. The only difference is the direction in which the data moves. A read moves data from outside the program into the program. A write moves data from within the program to some point outside the program, such as VDP RAM. The same registers are used for the same parameters. R0 is used for the VDP RAM address. R1 and R2 regulate the CPU RAM addresses. To use VMBR and VSBR, a REF directive must be included when using the Editor/Assembler, or the addresses of the routines must be EQUated to some two character label when using the Line-By-Line assembler. VSBR = >602C, VMBR = >6030. Here are the TMS9900 assembly language instructions to use the name data somewhere in the program:

```
NAME  BSS 30    SET ASIDE TEMP. STORAGE FOR NAME
      .
      .

LI    R0,288    LOAD R0 WITH THE VDP RAM ADDRESS
LI    R1,NAME   LOAD R1 WITH THE CPU RAM ADDRESS
LI    R2,30     LOAD R2 WITH THE DATA LENGTH
                (assuming fixed length of 30 bytes)
BLWP  @VMBR     PERFORM VDP MULTIPLE BYTE READ
```

Remember that all data looks the same to the computer. Everything is represented as a binary expression. When programming in assembly language, you must decide how data is to be interpreted. If a byte of memory contains the value >41, you must decide whether it means ASCII for "A", or if the value is to be treated as purely numeric (sixty five). INPUT X in TI BASIC will only let you enter a numeric string; anything else is rejected. In assembly language, you must provide for testing the input data to see if it is numeric and for rejecting it if it is not. Since the CURSOR routine will accept variable length data, you must also decide if variable length input is allowed or if the data must be fixed length.

Try retrieving a numeric string from the user at the keyboard. To simplify this example, it is required that the number be exactly four digits and a whole number or all zeroes.

```
PROMPT TEXT 'ENTER A 4 DIGIT NUMBER'   DEFINE PROMPT MESSAGE
NMTEST DATA >3Ø39       LABEL NMTEST CONTAINS >3Ø, ASCII FOR "Ø",
                              AND >39, ASCII FOR "9"
NUMBER BSS  4           TEMP. STORAGE FOR THE NUMBER
       .
       .
GETNUM LI   RØ,256      SCREEN ADDRESS FOR PROMPT - ROW 9, COL 1
       LI   R1,PROMPT   ADDRESS OF THE PROMPT
       LI   R2,22       LENGTH OF THE PROMPT
       BLWP @VMBW       DISPLAY THE PROMPT
       LI   RØ,288      SCREEN ADDRESS OF REPLY TO PROMPT
       LI   R1Ø,4       MAXIMUM LENGTH OF REPLY
       BL   @CURSOR     GET THE REPLY
       C    R7,4        IF ACTUAL LENGTH OF REPLY IS NOT 4,REPEAT
       JNE  GETNUM         THE PROMPT AND TRY AGAIN
       LI   RØ,288      ADDRESS OF REPLY
       LI   R1,NUMBER   WHERE TO PUT REPLY
       LI   R2,4        LENGTH OF REPLY
       BLWP @VMBR       READ THE REPLY FROM VDP RAM INTO CPU RAM
       CLR  R3          SET R3 TO ZERO
TEST   CB   @NUMBER(R3),@NMTEST COMPARE BYTE AT "NUMBER" PLUS
                         VALUE OF R3 TO BYTE AT ADDRESS NMTEST
                         1ST TIME THROUGH, R3=Ø THUS NUMBER+Ø.
                         BYTE AT NMTEST = >3Ø OR "Ø". IF BYTE
                         AT NUMBER + R3 IS LESS THAN >3Ø, IT
                         CANNOT BE A VALID ASCII NUMERIC. GO TO
       JLT  GETNUM       GETNUM, TRY AGAIN.
```

```
CB   @NUMBER(R3),@NMTEST+1 COMPARE BYTE AT NUMBER PLUS
                      R3 TO BYTE AT NMTEST+1. BYTE AT
                      NMTEST+1 = >39 OR "9". IF BYTE
                      AT NUMBER + R3 IS GREATER THAN >39, IT
JGT  GETNUM           CANNOT BE A VALID NUMERIC EITHER.
INC  R3               ADD 1 TO R3
CI   R3,R7            COMPARE R3 TO R7 (R7 CONTAINS 4)
JNE  TEST             IF R3 IS NOT EQUAL TO R7, THEN NOT
                      DONE. GO BACK AND PERFORM TEST LOOP
                      AGAIN.
```

If the check for a length of 4 is replaced by a check for a length of zero, then these instructions will work for variable length data. Example:

```
MOV   R7,R7
JEQ   GETNUM
```

Now a 4 digit string has been retrieved. The actual values in the string are ASCII codes for numerals. The sequence of the numeric symbols represent a decimal number. If you want to use the value of this response for any kind of arithmetic anywhere in the program, it must be converted to a binary value. Here is a sequence of TMS9900 assembly language instructions to do just that. This routine will only work for values up to 65,535 decimal, the maximum value of a word of memory. To use this routine in a program, place the numeric string into NUMBER, place the length of the string into R4, and perform a BL @CONVRT. The answer will be in R5 upon completion of the routine in binary integer format. If the number to be converted is too large, R5 is set to all zeroes. This routine assumes that you are passing a valid numeric string to it. Therefore, you must check for ASCII numeric symbols before performing this routine in order to get a meaningful result.

```
     DTEN     DATA >ØØØA
     NUMBER   BSS  6
     ••
     ••
     ••
Ø1   CONVRT   CLR  RØ
Ø2            CLR  R1
```

```
                CLR   R3
                CLR   R5
      MOVN      DEC   R4
                MOVB  @NUMBER(R4),R2
                SRL   R2,8
                AI    R2,->3Ø
                MOV   RØ,RØ
                JNE   EXP
                LI    RØ,1
                JMP   ACCUM
      EXP       MPY   @DTEN,RØ
                MOV   R1,RØ
                MPY   R1,R2
                MOV   R3,R2
      ACCUM     A     R2,R5
                JNO   NEXT
                CLR   R5
                RT
      NEXT      MOV   R4,R4
                JNE   MOVN
                RT
```

Here is how this routine performs the ASCII to binary conversion and some new instructions as well. To help in explaining the logic of the routine, assume that the number to be converted is decimal 234. Internally, the value at address "NUMBER" can be represented by a series of hexadecimal numbers each of which represents a byte. R4 contains a value of 3, the length of the string.

```
|    |    |    |    |
|>32 |>33 |>34 |>ØØ |
|____|____|____|____|
|    |    |    |
|    |    |    |_____NUMBER+3
|    |    |
|    |    |_____NUMBER+2
|    |
|    |_____NUMBER+1
|
|_____NUMBER+Ø
```

```
      CONVRT    CLR   RØ
                CLR   R1
                CLR   R3
                CLR   R5
      MOVN      DEC   R4
                MOVB  @NUMBER(R4),R2
                SRL   R2,8
                AI    R2,->3Ø
```

Lines 1 through 4 clear the registers which will be used in the routine. Line 5 DECrements (subtracts 1 from) R4. R4 contains the string length. Line 6 accesses the low order digit by using the base address of NUMBER plus the proper displacement. The string is comprised of 3 digits, and the displacement values of the digits going from high order to low order are 0, 1, 2. The displacement value for the last byte in a given series of bytes is always the number of bytes minus one, in this case, 3 - 1 = 2. "MOVN" is the label which will be used to create a loop. The first time through the loop, line 6 moves the low order byte (NUMBER plus the value in R4, NUMBER + 2) to R2. R2 now contains >3400. Line 7 performs a Shift Right Logical on R2 of 8 positions. R2 now contains >0034. Line 8 uses the Add Immediate instruction to strip off the >30 mask. By using an immediate value of −>30, the actual affect of this AI instruction is subtraction.

```
Ø9                 MOV  RØ,RØ
1Ø                 JNE  EXP
11                 LI   RØ,1
12                 JMP  ACCUM
```

Register 2 now contains >0004. Since the number sequence is a decimal number, you must multiply each digit by the power of ten which corresponds to its position in the sequence. The low order position of a decimal number represents units of 1. The first digit you have extracted multiplied by 1 would be equal to itself. Therefore, the low order digit can be used as it is. Line 9 moves R0 to itself so that line 10 can check R0 for a value of zero. JNE stands for Jump if Not Equal. The first time through, R0 is equal to zero, and the JNE instruction at line 10 has no effect. Line 11 Loads Immediate R0 with a value of 1. Line 12 performs an unconditional Jump to the label ACCUM.

```
17   ACCUM      A    R2,R5
18              JNO  NEXT
19              CLR  R5
2Ø              RT
21   NEXT       MOV  R4,R4
22              JNE  MOVN
23              RT
```

Continuing with the first time through, line 17, ACCUM, Adds the contents of R2 to R5. Should the value in R5 (which is being used as an accumulator for this routine) become too great to fit in R5, the overflow bit will be set in the status register. Line 18 checks for this condition with the Jump if No Overflow instruction. As long as the overflow bit is not set, instruction logic continues at the label NEXT. If the overflow bit is set, then the next two instructions clear R5 and return to the calling program. At line 21, NEXT moves R4 to itself. Line 22 checks R4 for a value of zero by testing the equal bit in the status register. If R4 is equal to zero at this point in the logic of the subroutine, the subroutine's task is done and line 23 returns to the calling program address. The first time through, R4 is equal to 2, and the routine Jumps to the label MOVN.

The second pass through the routine takes a slightly different course. R4 is DECremented to 1. Line 6 moves the next byte of the number (>33) to R2. R2 is shifted right 8 positions to become >0033, and the >30 mask is removed giving >0003. R0 now contains 1, so line 10 triggers a Jump to the label EXP at line 13.

```
13    EXP      MPY  @DTEN,RØ
14             MOV  R1,RØ
15             MPY  R1,R2
16             MOV  R3,R2
```

Line 13 uses the multiply (MPY) instruction to calculate the power of ten which corresponds to the relative position of the decimal digit. The MPY instruction multiplies the first and second operands (which must name a register). The MPY instruction uses two successive registers just as the DIV instruction does. Like DIV, usage of the second register is implied, meaning that the additional register is not specified anywhere in the instruction. In the example, DTEN is multiplied by the contents of R0, and the result ends up in R1 because R1 is the next register after R0. Now examine the contents of the locations involved.

```
Before the MPY:          DTEN        RØ         R1
                         -----      -----      -----

                         >ØØØA      >ØØØ1      >ØØØØ


After the MPY:

                         >ØØØA      >ØØØØ      >ØØØA
```

Should the result of an MPY instruction be larger than one word, the result will continue on into the named register. In this example, register 0 would contain the most significant bits and register 1 would contain the least significant bits. The MPY instruction does not affect the status register.

Line 14 moves the answer of ten, which is in R1, into R0. This is done to prepare R0 for the next loop. On each pass through the loop, R0 will be mutiplied by ten. In this way, the contents of R1 will be equal to the power of ten needed for each decimal position (1, 10, 100, 1000, etc). Remember that a move merely copies the contents of one location to another. The contents of R1 are still intact. At line 15, the value in R2, which is the numeral being converted (>0003) is multiplied by the value in R1 (the power of ten for the position of this numeral.) Here is the effect of the MPY instruction at line 15.

```
                        R1          R2         R3
Before the MPY:        -----       -----      -----
                       >ØØØA       >ØØØ3      >ØØØØ


After  the MPY:        >ØØØA       >ØØØØ      >ØØ1E
```

The result ends up in R3, the next available register after R2, which was the register named in the second operand of the MPY instruction. Line 16 moves the answer to R2. This is done because line 17, ACCUM, expects the value that is to be added to R5 to be in R2.

The loop is repeated one more time to extract the high order digit. This digit represents units of 10 or 100. The number extracted is 2. Multiplied by 100, it equals 200. Adding it to R5 by ACCUM brings the total value in R5 to >00EA, or 234 decimal. The value in R5 can now be used for any arithmetic that might be needed.

You do not need to use an extensive routine like CURSOR in your program to interact with keyboard input or to make good use of a utility like KSCAN. In the first two program examples, the return address was altered in order to allow you to view the results of the program. Now that you know about KSCAN, here is a more proper way to end a program. The return address is unaltered, and in order to create a pause, KSCAN is used to detect the pressing of any key. The program will wait until some key is pressed and then end. Here are the program segments to do this:

```
EOJ    CLR    @STATUS      CLEAR THE GPL STATUS BYTE
SCAN   BLWP   @KSCAN       PERFORM KEYBOARD SCAN
       MOVB   @STATUS,@STATUS COMPARE THE GPL STATUS BYTE TO >ØØ
       JEQ    SCAN         IF NO KEY WAS PRESSED, SCAN AGAIN
       MOV    @SAVRTN,R11 MOVE RETURN ADDRESS TO R11
       CLR    @STATUS      CLEAR THE GPL STATUS BYTE
       RT                  RETURN VIA R11
```

Another application of KSCAN would be the detection of special key stroke values like Clear or Quit. Remember that while your TMS9900 assembly language program is running, it is in complete control of the computer. In order for the computer user to abort the program, the program must provide for the detection of such a command. Since these types of key strokes do not represent any displayable data, there is no need for a routine like CURSOR. Depending on how you program the actions to be taken for the various control values, your computer can respond to these commands accordingly. Here are the program segments to detect Quit (FCTN =).

```
QUITV BYTE   >Ø5
•
•
SCAN  CLR    @STATUS    CLEAR THE GPL STATUS BYTE
      BLWP   @KSCAN     PERFORM KEYBOARD SCAN
      MOVB   @STATUS,@STATUS SEE IF A KEY HAS BEEN PRESSED
      JEQ    SCAN       IF NOT, SCAN AGAIN
      CB     @QUITV,@KEYVAL SEE IF "QUIT" WAS PRESSED
      JEQ    ABORT      IF IT WAS, GO TO END OF JOB
      RT                IF NOT, RETURN
```

Important: Before you abort a program, make sure you close any opened files and clean up any other "loose ends."

The usual response to pressing Quit while operating under TI BASIC, Extended BASIC and most utilities is for the computer to return to the main title screen. Here are the program segments to accomplish this.

```
GPLWS   EQU   >83EØ
•
•
ABORT   LIMI  2        ENABLE INTERUPTS
        LWPI  GPLWS    LOAD GPL WORKSPACE REGISTERS
        BLWP  @>ØØØØ   BRANCH THROUGH THE VECTOR >ØØØØ
```

The LIMI instruction stands for Load Interrupt Mask Immediate and is used to enable/disable an interrupt. LIMI 0 (interrupts disabled) is the normal state of the computer. The instruction places the least significant 4 bits of the contents of the immediate operand in the interrupt mask of the status register. Without interrupts, the CPU processes one instruction or piece of data after another. This processing sequence occurs at a steady pulse. Certain operations require that you interrupt this orderly process, usually to allow for differences in speed between the various microprocessors which make up the computer. LIMI 2 enables interrupts at levels 0, 1, and 2. Since the branch through >0000 returns to the main title screen by way of the GPL resident routine, it is necessary to have the WP register pointing to the workspace used by GPL.

Here is a chart which gives the key stroke values for the FCTN key and numeric key combinations.

		KEY STROKE VALUE	
COMBINATION	NAME	DECIMAL	HEXADECIMAL
FCTN 1	DELETE	Ø3	>Ø3
FCTN 2	INSERT	Ø4	>Ø4
FCTN 3	ERASE	Ø7	>Ø7
FCTN 4	CLEAR	Ø2	>Ø2
FCTN 5	BEGIN	14	>ØE
FCTN 6	PROCEED	12	>ØC
FCTN 7	AID	Ø1	>Ø1
FCTN 8	REDO	Ø6	>Ø6
FCTN 9	BACK	15	>ØF
FCTN Ø		188	>BC
FCTN =	QUIT	Ø5	>Ø5

Here is a sample program which demonstrates some of the KSCAN principles. The border graphics and prompts use the same logic as the sample program in Chapter Seven. When the program is run, any key you press will result in the character being displayed (providing it is displayable), and the decimal value for the key or key combination is displayed also. The next prompt will only accept REDO or ESCAPE. REDO repeats the entire sequence, and ESCAPE will return to the main title screen. You can use this program to find the value for any key stroke or combination.

The routine FIGUR uses the same logic as the first sample program to convert binary to displayable ASCII. FIGUR uses a loop to allow it to process any number that will fit into a single register. FIGUR handles both the conversion and the display of the value.

```
*KEYBOARD INPUT PROGRAM EXAMPLE*
*EDITOR/ASSEMBLER VERSION      *

        DEF  GO                             DEFINE THE ENTRY POINT
        REF  VWTR,VSBW,VMBW,KSCAN           REF ROUTINES TO BE USED
WR      BSS  >20                            SET ASIDE ALT. WORKSPACE
STATUS  EQU  >837C                          GPL STATUS BYTE
KEYADR  EQU  >8374                          KEYBOARD DEVICE ADDRESS
KEYVAL  EQU  >8375                          KEYSTROKE VALUE ADDRESS
DTEN    DATA >A                             DECIMAL TEN
BORDER  DATA >FFFF,>2020,>2020,>2020        DEFINE BORDER PATTERN
        DATA >2020,>2020,>2020,>2020
        DATA >2020,>2020,>2020,>2020
        DATA >2020,>2020,>2020,>FFFF
MSG1    TEXT '** PRESS ANY KEY     **'      DEFINE 1ST PROMPT
MSG2    TEXT '* KEYSTROKE VALUE IS *'       DEFINE 2ND PROMPT
MSG3    TEXT '* PRESS REDO/ESCAPE  *'       DEFINE 3RD PROMPT
REDOV   BYTE >06                            "REDO" VALUE
ESCPV   BYTE >0F                            "ESCAPE" KEY VALUE
SAV11   BSS  2
GO      MOV  R11,@SAV11             SAVE RETURN ADDRESS
        LWPI WR                     LOAD WORKSPACE POINTER IMMEDIATE
        LI   R0,>070D               SET BACKGROUND BORDER TO MAGENTA
        BLWP @VWTR
        LI   R0,>039F               SET BORDER CHARACTER TO MAGENTA
        LI   R1,>DD00
        BLWP @VSBW
        LI   R0,>380                SET CHARACTERS TO BLACK ON WHITE
        LI   R1,>1F00
CLOOP   BLWP @VSBW
        CI   R0,>039E
        JEQ  BPUT
        INC  R0
        JMP  CLOOP
BPUT    LI   R0,0                   LOAD BORDER PATTERN
        LI   R1,BORDER
        LI   R2,32
BLOOP   BLWP @VMBW
        CI   R0,736
        JEQ  EXIT
        AI   R0,32
        JMP  BLOOP
EXIT    LI   R0,261                 PUT UP 1ST PROMPT
        LI   R1,MSG1
        LI   R2,22
        BLWP @VMBW
        CLR   @KEYADR            CLEAR KEYADR & KEYVAL
SCAN1   CLR   @STATUS            CLEAR GPL STATUS BYTE
        BLWP  @KSCAN             PERFORM KSCAN
        MOVB  @STATUS,@STATUS    SEE IF ANY KEY WAS PRESSED
        JEQ   SCAN1              IF NOT, SCAN AGAIN
        LI    RØ,325             PUT UP 2ND PROMPT
        LI    R1,MSG2
        BLWP  @VMBW
        LI    RØ,395             DISPLAY THE KEYED CHARACTER
        MOVB  @KEYVAL,R1
        BLWP  @VSBW
        MOVB  @KEYVAL,R4
        SRL   R4,8
        LI    R3,4Ø4             DISPLAY THE DECIMAL VALUE OF THE
        LI    RØ,4Ø6             CHARACTER
        BL    @FIGUR
        LI    RØ,485             PUT UP 3RD PROMPT
        LI    R1,MSG3
        LI    R2,22
        BLWP  @VMBW
SCAN2   CLR   @STATUS            PERFORM KSCAN AGAIN
        BLWP  @KSCAN
        MOVB  @STATUS,@STATUS
        JEQ   SCAN2
        CB    @KEYVAL,@ESCPV     ESCAPE OR REDO PRESSED?
        JEQ   ESCAP              IF ESCAPE, GO TO ESCAP
        CB    @KEYVAL,@REDOV
        JNE   SCAN2
        B     @BPUT              IF REDO, GO TO BPUT
FIGUR   MOV   R4,R5              ROUTINE TO CONVERT INTERNAL VALUE
        CLR   R4                 TO DISPLAYABLE DECIMAL NUMERICS
        DIV   @DTEN,R4           INPUT:R4=VALUE,R3=1ST SCREEN
        AI    R5,>3Ø              ADDRESS OF ANSWER,RØ=LAST SCREEN
        SLA   R5,8                ADDRESS
        MOV   R5,R1
        BLWP  @VSBW
        DEC   RØ
        C     RØ,R3
        JHE   FIGUR
        RT
ESCAP   CLR   @STATUS            CLEAR THE STATUS BYTE
        MOV   @SAV11,R11         RETURN
        RT
        END
```

To try out this program using the Line-by-Line assembler, observe the coding differences previously outlined for the Editor/Assembler and Line-by-Line assembler products in Chapter Five. Here is the listing for use with the Line-by-Line assembler:

```
* KEYBOARD INPUT PROGRAM EXAMPLE *
* MINI-MEMORY VERSION            *
*                                *
        AORG >7DØC
VW      EQU  >6Ø34
VS      EQU  >6Ø24
VM      EQU  >6Ø28
KS      EQU  >6Ø2Ø
WR      BSS  >2Ø
DT      DATA >A
BD      DATA >8Ø8Ø,>2Ø2Ø,>2Ø2Ø,>2Ø2Ø
        DATA >2Ø2Ø,>2Ø2Ø,>2Ø2Ø,>2Ø2Ø
        DATA >2Ø2Ø,>2Ø2Ø,>2Ø2Ø,>2Ø2Ø
        DATA >2Ø2Ø,>2Ø2Ø,>2Ø2Ø,>8Ø8Ø
M1      TEXT '** PRESS ANY KEY    **'
M2      TEXT '* KEYSTROKE VALUE IS *'
M3      TEXT '* PRESS REDO/ESCAPE  *'
EV      DATA >ØFØØ
RV      DATA >Ø6ØØ
GO      MOV  R11,R1Ø
        LWPI WR
        LI   RØ,>Ø7Ø6
        BLWP @VW
        LI   RØ,>39Ø
        LI   R1,>66ØØ
        BLWP @VS
        LI   RØ,>383
        LI   R1,>1FØØ
CL      BLWP @VS
        CI   RØ,>38F
        JEQ  BP
        INC  RØ
        JMP  CL
BP      LI   RØ,Ø
        LI   R1,BD
        LI   R2,32
BL      BLWP @VM
        CI   RØ,736
        JEQ  EX
        AI   RØ,32
        JMP  BL
EX      LI   RØ,261           PUT UP 1ST PROMPT
        LI   R1,M1
        LI   R2,22
        BLWP @VM
S1      CLR  @>837C           CLEAR GPL STATUS BYTE
        BLWP @KS  N           PERFORM KSCAN
        MOVB @>837C,@>837C    SEE IF ANY KEY WAS PRESSED
        JEQ  S1               IF NOT, SCAN AGAIN
        LI   RØ,325           PUT UP 2ND PROMPT
        LI   R1,M2
        BLWP @VM
        LI   RØ,395           DISPLAY THE KEYED CHARACTER
        MOVB @>8375,R1
        BLWP @VS
        CLR  R4
        MOVB @>8375,R4
        SRL  R4,8
        LI   R3,4Ø4           DISPLAY THE DECIMAL VALUE OF THE
        LI   RØ,4Ø6             CHARACTER
        BL   @FG
        LI   RØ,485           PUT UP 3RD PROMPT
        LI   R1,M3
        LI   R2,22
        BLWP @VM
S2      CLR  @>837C           PERFORM KSCAN AGAIN
        BLWP @KS
        MOVB @>837C,@>837C
        JEQ  S2
        CB   @>8375,@EV       "ESCAPE" OR "REDO" ?
        JEQ  ES               IF ESCAPE, GO TO ESCAP
        CB   @>8375,@RV
        JNE  S2
        B    @BP              IF REDO, GO TO BPUT
FG      MOV  R4,R5            ROUTINE TO CONVERT INTERNAL VALUE
        CLR  R4               TO DISPLAYABLE DECIMAL NUMERICS
        DIV  @DT,R4           INPUT:R4=VALUE,R3=1ST SCREEN ADDRESS
        AI   R5,>3Ø            OF ANSWER,RØ=LAST SCREEN ADDRESS
        SLA  R5,8
        MOV  R5,R1
        BLWP @VS
        DEC  RØ
        C    RØ,R3
        JHE  FG
        B    *11
ES      CLR  @>837C
        MOV  R1Ø,R11
        B    *11
        END
```

EDITOR/ASSEMBLER MANUAL REFERENCES

The following references will provide you with some more information on processing keyboard input.

Read: Section 10.2 page 164
Section 16.2 page 250
Section 16.3 page 264 through Section 16.3.1 page 264
Section 24.11 page 440 through Section 24.11.3 page 442

Look up these terms in the glossary:

ASCII
Console
Interrupt Mask Bits

CHAPTER NINE FILE HANDLING

The creation, reading, and updating of data files through TMS9900 assembly language is another important function which involves VDP RAM. File specifications describing a file's record length, length format, data and file format, and mode of operation are accumulated in a block of memory known as a Peripheral Access Block (PAB). The PAB details all the information required by the computer to recognize and access the files you want. The reading, writing and updating of file data is handled by resident routines called DSRs, Device Service Routines. File management is implemented through TMS9900 assembly language by manipulating data within the PAB for a file and making the PAB data available to the correct DSR.

When programming in TI BASIC, you supplied all of the parameters about a file in the OPEN statement, which opened the file and specified how the file was to be used, input, output, update or append. The file was then accessed within the program by INPUT or PRINT statements. When you were finished with the file, it needed to be closed. These four file functions, defining the file, opening the file, reading from or writing to the file, and closing the file, are always required of any programming language. TMS9900 assembly language requires a few more lines of code to accomplish the same result than TI BASIC does, but file handling generally is not very complex.

The first step is to define the file characteristics to the computer. The bytes of data which make up the PAB define the key parameters of the file. The actual number of bytes which make up a PAB is variable, depending on the device/file name selected. The first byte (the zero byte) of the PAB instructs the Device Service Routine as to what operation you wish to perform (open, read, write, close, etc.) The initial setting of this byte for most files would be >00, or open. Here are the available op-codes for PAB byte zero.

```
VALUE       OPERATION
=====================

 >ØØ        OPEN
 >Ø1        CLOSE
 >Ø2        READ
 >Ø3        WRITE
 >Ø4        RESTORE/REWIND
 >Ø5        LOAD
 >Ø6        SAVE
 >Ø7        DELETE FILE
 >Ø9        STATUS
```

Op-code >08 (scratch record) is generally not used by the TI 99/4 or 99/4A because the disk controller does not allow a record to be scratched.

The next PAB byte (the 1 byte) controls several functions. Depending on which bits are set on or off, it defines the file's open mode (input, output, update), record type (fixed, or variable length), the type of data to be contained (display or internal format), and whether the file is to be processed sequentially or by random access (relative files). All of these parameters are defined by various combinations of bits 3 through 7 of the 1 byte. Bits 0,1,and 2 are used to report various error conditions as they occur. The initial setting of bits 0,1, and 2 should always be zeroes (no errors). Here is a summary of PAB byte 1 values and their meanings.

PAB BYTE 1 VALUES AND MEANINGS

RELATIVE FILES - ALL RELATIVE FILES ARE FIXED LENGTH	
>Ø1	UPDATE, DISPLAY
>Ø3	OUTPUT, DISPLAY
>Ø5	INPUT, DISPLAY
>Ø9	UPDATE, INTERNAL
>ØB	OUTPUT, INTERNAL
>ØD	INPUT, INTERNAL
SEQUENTIAL FILES	
>Ø2	OUTPUT, FIXED, DISPLAY
>Ø4	INPUT, FIXED, DISPLAY
>Ø6	APPEND, FIXED, DISPLAY
>ØA	OUTPUT, FIXED, INTERNAL
>ØC	INPUT, FIXED, INTERNAL
>ØE	APPEND, FIXED, INTERNAL
>12	OUTPUT, VARIABLE, DISPLAY
>14	INPUT, VARIABLE, DISPLAY
>16	APPEND, VARIABLE, DISPLAY
>1A	OUTPUT, VARIABLE, INTERNAL
>1C	INPUT, VARIABLE, INTERNAL
>1E	APPEND, VARIABLE, INTERNAL

Bytes 2,3 (1 word) contain the address in VDP RAM which is to be used as a buffer (temporary storage space) for each record as it is read or written. Byte 4 defines the logical record length in bytes. For variable length records this value is the maximum length. The largest value that can be defined using one byte is >FF or 255. Byte 5 defines the number of bytes to be written for a write operation, or the actual number of bytes read for a read operation. For fixed length records PAB byte 4 and 5 should be set equal when writing, and will always be equal when reading. For variable length records PAB byte 5 can be tested to determine the actual record length on a read and can be dynamically changed for each write. The value in PAB byte 5 can never be greater than the logical, or maximum record length. PAB bytes 6,7 (1 word) are only used for relative (random access) files. This word contains the relative record number to be accessed. The most significant bit of this word is ignored so that the range of possible values is from zero (the first record on a relative file) to 32,767. Byte 8 is only used for files to be stored on a cassette tape device. The value in this byte is the amount of screen offset (>60 in BASIC, >00 in assembly). The cassette DSR needs this value for the screen prompts it must display for operation of the cassette recorder. Byte 9 indicates the length of the file descriptor which begins in byte 10. The file description can be of variable length. This is where you would place the file/device name you have chosen for the file ("CS1", "DSK1.FILE", "RS232.BA=300", etc.). Since the actual length of this entry will vary depending on the device selected, the computer needs the length of this data in byte 9. Here is a PAB as it would be coded in a TMS9900 assembly program:

```
                  ____________________________OPERATION: >ØØ = OPEN
                  |
                  |  _________________________FILE TYPE: >12 = OUTPUT,
                  |  |                        VARIABLE, DISPLAY,
                  |  |                        SEQUENTIAL
                  |  |
                  |  |    ____________________BUFFER ADDRESS IN VDP RAM
                  |  |    |                   (SYMBOLIC OR ACTUAL)
                  |  |    |
                  |  |    |
                  |  |    |         __________RECORD LENGTH: >5Ø (8Ø)
                  |  |    |         |
                  |  |    |         |  _______CHARACTER COUNT
                  |  |    |         |  |
                  |  |    |         |  |    __RELATIVE RECORD NUMBER
                  |  |    |         |  |    |
                  |  |    |         |  |    |     _CASSETTE SCREEN OFFSET
                  |  |    |         |  |    |     T
                  |  |    |         |  |    |     |  _FILE DESCRIPTOR
                  |  |    |         |  |    |     |  |¯LENGTH
PAB      DATA    >ØØ12,BUFADR,>5ØØØ,>ØØØØ,>ØØØA

                  ____________________________FILE DESCRIPTOR
                  |
                  |                           THIS DESCRIPTOR IS >ØA
                  |                           (1Ø) CHARACTERS LONG
         TEXT    'DSK1.FILE1'
```

PABs are coded in your program and then placed into VDP RAM with a routine such as VMBW. VDP RAM is used by all DSRs for PABs and buffer space. Since there are many important tables and other data in VDP RAM, only certain areas should be used for PABs and buffers. The first free address in VDP RAM normally used for PABs is >F80. This address actually overlaps the end of the pattern descriptor table, but the character codes >F0 through >FF are not defined as displayable, so this usually does not cause any problems. VDP RAM free space extends through VDP RAM address >37D6. This represents a considerable amount of space for PAB and buffer needs. Be careful to use this space for only these functions. Where multiple files are to be used, enough space must be allocated between PABs and buffers to insure file data integrity. Here are the program segments which establish a PAB and buffer.

```
BUFADR     EQU    >1ØØØ  VDP RAM ADDRESS FOR RECORD BUFFER
PABADR     EQU    >ØF8Ø  VDP RAM ADDRESS FOR PAB
PAB        DATA   >ØØ12,BUFADR,>5ØØØ,>ØØØØ,>ØØØ9
           TEXT   'DSK1.FILE '
           •
           •

           LI     RØ,PABADR
           LI     R1,PAB
           LI     R2,2Ø
           BLWP   @VMBW
```

Once the PAB and buffer for a file have been established, the actual access is accomplished via the Device Service Routine (DSR). By pointing to the PAB (file definition) to be accessed, manipulating byte zero of the PAB (operation), and branching to the DSR, the file can be opened, read from, written to, closed, etc. The DSR for all peripherals except cassette is accessed by including a REF DSRLNK in your program when using the Editor/Assembler, or by EQUating the address >6038 to some 2 character label when using the Line-by-Line assembler. The cassette DSR is a GPL (Graphics Programming Language) routine located in GROM (Graphics Read Only Memory). Cassette file handling and DSRs will be covered later on. The pointer needed by DSRLNK is the address of the file descriptor byte. This value must be placed in the word at address >8356. For the above example:

```
           LI     R6,PAB+9
           MOV    R6,@>8356
```

DSRLNK is then invoked with a BLWP instruction. DSRLNK also needs the value 8 passed to it to complete the instruction.

```
           BLWP   @DSRLNK
           DATA   8
```

The same DSRLNK instruction is used for any operation on any file except cassettes. The actual operation performed on the file by each BLWP to DSRLNK (open, read, etc.) depends on the value in PAB byte zero. The file to be accessed depends on the value you place at >8356. DSRLNK will determine if the file characteristics match the device, and if the operation requested is compatible with the device/file characteristics. Errors of this kind as well as end of file (disk), and other processing conditions are detected by DSRLNK and are reported in BITS 0,1,2 of PAB byte 1. Assuming no errors occur during file processing, DSRLNK handles all aspects of the process, such as updating the catalog entries for a disk file. Your TMS9900 assembly language program must check for error conditions and provide for some action to be taken as a result. When errors do occur, the equal bit in the status register is set (1). If no errors occur, the equal bit in the status register is reset (0).

If the device you have selected is either RS232 or TP (Thermal Printer), you must save the GROM read and GROM write addresses before each BLWP @DSRLNK, and restore them afterwards. The DSRs for these devices render these addresses indeterminate. Here are the program segments to save and restore these addresses.

```
        REF   GRMRA
        REF   GRMWA
        •
        •
SAVADR  BSS 2
        •
        •
        MOVB @GRMRA,@SAVADR     GET FIRST BYTE OF ADDRESS
        NOP
        MOVB @GRMRA,@SAVADR+1   GET SECOND BYTE
        DEC  @SAVADR            DECREMENT THE ADDRESS
        •
        •
        BLWP @DSRLNK            ACCESS PERIPHERAL
        DATA 8
        •
        •
        MOVB @SAVADR,@GRMWA     RESTORE FIRST BYTE OF ADDRESS
        NOP
        MOVB @SAVADR+1,@GRMWA   RESTORE SECOND BYTE OF ADDRESS
```

In the above example, the pseudo instruction "NOP" is used to allow a time delay between accesses to the GROM addresses. NOP performs no function, but it takes up as much time as a real instruction would. NOPs can be useful in this way when there are timing considerations which dictate that your TMS9900 assembly program idle while a slower area of the computer catches up.

Here is a program example of sequential file access. The input file contains variable length 80 character records in display format. Each record contains a first name with a maximum length of 14 characters and a last name. The actual length of the last name field (and therefore the record) is variable, although the first character of last name must begin in position 15. This program reads the file of names and selects the third record on the file. The first and last names of the third record are then displayed.

```
        DEF  BEGIN
        REF  DSRLNK,VSBW,VMBW,VMBR,VSBR
STATUS  EQU  >837C        GPL STATUS BYTE ADDRESS
POINTR  EQU  >8356        DSR POINTER ADDRESS
BUFADR  EQU  >1000        VDP RAM ADDRESS FOR RECORD BUFFER
PABADR  EQU  >F80         VDP RAM ADDRESS FOR PAB
READ    BYTE >02          "READ"  OP-CODE
CLOSE   BYTE >01          "CLOSE" OP-CODE
EOF     DATA 0            END OF FILE FLAG
```

```
PAB     DATA >ØØ14,BUFADR,>5ØØØ,>ØØØØ,>ØØØA   PAB DATA
        TEXT 'DSK2.FILE1'
ERRMSG  TEXT 'I/O ERROR='     DSR ERROR MESSAGE
CPUBUF  BSS  8Ø               CPU RAM RECORD BUFFER ADDRESS
FNAME   EQU  CPUBUF           FIRST NAME ADDRESS
LNAME   EQU  CPUBUF+14        LAST NAME ADDRESS
LEN     BSS  2                ACTUAL RECORD LENGTH WORKSPACE
RETURN  BSS  2                SAVE RETURN ADDRESS AREA
WR      BSS  >2Ø              WORKSPACE REGISTERS
BEGIN   MOV  R11,@RETURN      SAVE RETURN ADDRESS
        LWPI WR               LOAD WORKSPACE POINTER
        LI   RØ,PABADR        VDP RAM ADDRESS FOR PAB
        LI   R1,PAB           CPU RAM ADDRESS OF PAB DATA
        LI   R2,2Ø            LENGTH OF DATA
        BLWP @VMBW            WRITE PAB TO VDP RAM
        BL   @DSR             OPEN THE FILE
        MOVB @READ,R1         LOAD READ OP-CODE INTO R1
        LI   RØ,PABADR        LOAD PAB ADDRESS INTO RØ
        BLWP @VSBW            PUT READ INTO PAB BYTE Ø
        CLR  R4               CLEAR RECORD COUNTER
READF   BL   @DSR             PERFORM DSR ROUTINE
        MOV  @EOF,@EOF        CHECK FOR END OF FILE
        JNE  EOJ              IF EOF GO TO END OF JOB
        INC  R4               ADD 1 TO RECORD COUNT
        CI   R4,3             CHECK FOR THIRD RECORD
        JNE  READF            IF NOT THIRD, READ AGAIN
        LI   RØ,PABADR+5      ADDRESS OF CHARACTER COUNT
        BLWP @VSBR            READ COUNT INTO LEFT BYTE R1
        SRL  R1,8             SHIFT LEFT TO RIGHT
        MOV  R1,R2            MOVE VALUE TO R2
        MOV  R1,@LEN          SAVE VALUE IN LEN
        LI   RØ,BUFADR        VDP RAM RECORD BUFFER ADDRESS
        LI   R1,CPUBUF        CPU RAM ADDRESS FOR RECORD
        BLWP @VMBR            GET RECORD FROM VDP TO CPU RAM
        LI   RØ,29Ø           SCREEN ADDRESS FOR FIRST NAME
        LI   R1,FNAME         CPU RAM ADDRESS OF FIRST NAME
        LI   R2,14            LENGTH OF FIRST NAME FIELD
        BLWP @VMBW            DISPLAY FIRST NAME
        LI   RØ,3Ø5           SCREEN ADDRESS FOR LAST NAME
        LI   R1,LNAME         CPU RAM ADDRESS OF LAST NAME
        MOV  @LEN,R2          MOVE RECORD LENGTH INTO R2
        AI   R2,-14           SUBTRACT LENGTH OF FIRST NAME
*                             DIFFERENCE IS LAST NAME LENGTH
        BLWP @VMBW            DISPLAY LAST NAME
        JMP  EOJ              GO TO END OF JOB
DSR     LI   R6,PABADR+9      LOAD R6 WITH DESCRIPTOR LENGTH
        MOV  R6,@POINTR       MOVE ADDRESS TO POINTER
        BLWP @DSRLNK          PERFORM DSRLNK
        DATA 8                DATA NEEDED BY DSRLNK
*
```

```
ØØ59         JEQ  DSRERR         CHECK FOR ERROR
ØØ6Ø         RT                  RETURN
ØØ61 DSRERR  INC  @EOF           SET EOF INDICATOR
ØØ62         LI   RØ,PABADR+1    ADDRESS OF PAB BYTE 1
ØØ63         BLWP @VSBR          READ PAB BYTE 1 INTO R1
ØØ64         SRL  R1,13          SHIFT HIGH ORDER 3 BITS TO LOW
ØØ65         CI   R1,5           CHECK FOR EOF VALUE=5
ØØ66         JNE  IOERR          IF NOT EOF THEN OTHER ERROR
ØØ67         RT                  IF EOF THEN RETURN
ØØ68 IOERR   AI   R1,>3Ø         MASK ERROR CODE
ØØ69         SLA  R1,8           SWAP LOW ORDER TO HIGH ORDER
ØØ7Ø         LI   RØ,299         DISPLAY ERROR CODE
ØØ71         BLWP @VSBW          ON THE SCREEN
ØØ72         LI   RØ,288         DISPLAY ERROR MESSAGE
ØØ73         LI   R1,ERRMSG
ØØ74         LI   R2,1Ø
ØØ75         BLWP @VMBW
ØØ76 EOJ     MOV  @EOF,@EOF      IF EOF REACHED, DSR WILL
ØØ77         JNE  NOCLOS         CLOSE FILE
ØØ78         MOVB @CLOSE,R1      MOVE CLOSE OP-CODE TO R1
ØØ79         LI   RØ,PABADR      LOAD PAB ADDRESS
ØØ8Ø         BLWP @VSBW          WRITE CLOSE OP-CODE TO PAB Ø
ØØ81         BL   @DSR           CLOSE FILE
ØØ82 NOCLOS  DECT @RETURN        ALTER RETURN ADDRESS
ØØ83         MOV  @RETURN,11     MOVE RETURN ADDRESS INTO R11
ØØ84         RT                  RETURN
ØØ85         END
```

Line 25 opens the file because the op-code in PAB byte zero is originally set at >00 (the op-code for "OPEN"). Lines 26,27, and 28 write the op-code for "READ" (>02) to PAB byte zero. Once this is done, each successive DSRLNK performs a read and will continue to perform reads until the op-code is changed to some other value. The subroutine "DSR" at lines 54 through 60 contain the instructions and data which perform the DSR. Line 59 uses the Jump if EQual instruction to test the equal bit in the status register. DSRLNK sets this bit if there are any errors. End of file is reported as an error with a value of 5 in bits 0,1,2 of PAB byte 1. Lines 61 through 67 test for end of file. If end of file has been reached, then the word "EOF" is changed from zero to 1. DSRLNK will close the file for you if end of file has been reached. For this program example, end of file would only occur if the file contained fewer than 3 records. If the condition reported is something besides end of file, lines 68 through 75 display the error code and an error message.

Lines 30 through 53 detail the actions which read through the file until the third record is found. Lines 36 and 37 extract the character count from PAB byte 5. VSBR (VDP Single Byte Read) reads from the address in R0 into the left byte of R1. The value from PAB byte 5 is the actual length of the record just read. This value is used along with VMBR to get the record from its VDP RAM buffer into a buffer which the program can access directly (lines 39 through 43). Lines 44 through 47 get and display the first name. Lines 48 through 52 get and display the last name. The calculation at line 51 determines the actual length of the last name field by subtracting the length of the first name field from the actual record length which was found in PAB byte 5.

At end of job (line 76) a determination must be made as to whether or not the file needs to be closed. To close the file, the op-code for "CLOSE" (>01)is written to PAB byte zero and one more access to DSRLNK is performed. Then the program follows previous examples which wait for some key to be pressed before ending. Here is a TI BASIC program which will create a name file like the one used as input in the TMS9900 assembly language program example.

```
100 CALL CLEAR
110 OPEN #2:"DSK2.FILE1",OUTPUT,VARIABLE 80
120 INPUT "ENTER AN E WHEN DONE":X$
130 IF X$="E" THEN 190
140 INPUT "FIRST NAME? ":FN$
150 IF LEN(FN$)>14 THEN 140
160 INPUT "LAST NAME? ":LN$
170 PRINT #2:FN$,LN$
180 GOTO 120
190 CLOSE #2
200 END
```

Here is a hexadecimal display of the file created by the TI BASIC program. This was produced with the TI Programming Aids II diskette. While the BASIC program limits the size of the first name field to a maximum of 14 bytes, there is no such specification governing the size of the last name. If you intend for a TMS9900 assembly language program to process a file created by a TI BASIC program, it is important that the TI BASIC program be coded to specify definite field positions and field lengths. Or, as was done in this case, you can use a dump utility such as the one in TI Programming Aids II to reveal how the file's records will appear to the assembly program.

```
FILE TYPE IS DISPLAY

RECORD TYPE IS VARIABLE 8Ø

13 44 41 56        (.DAV)
49 44 2Ø 2Ø        (ID  )
2Ø 2Ø 2Ø 2Ø        (    )
2Ø 2Ø 2Ø 53        (   S)
54 4F 4E 45        (TONE)
14 4D 41 52        (.MAR)
56 49 4E 2Ø        (VIN )
2Ø 2Ø 2Ø 2Ø        (    )
2Ø 2Ø 2Ø 53        (   S)
5Ø 41 52 4B        (PARK)
53 15 57 41        (S.WA)
59 4E 45 2Ø        (YNE )
2Ø 2Ø 2Ø 2Ø        (    )
2Ø 2Ø 2Ø 2Ø        (    )
4E 45 57 43        (NEWC)
4F 4D 45           (OME )
```

When creating files with TMS9900 assembly programs you must design the layout of each record. You decide how long each field can be and its beginning position within the record. One of the inefficiencies TI BASIC is that it assigns string space within records in predetermined blocks of bytes. Unless you code the TI BASIC program to structure each field, TI BASIC follows a preset algorithm for field lengths. Alphanumeric strings, for example, always start out with a length of 14 bytes. In the case of the first name field, even if all the first names entered are never longer than 5 bytes, 14 byte strings are built for each. Discounting line 150, which restricts the field length, if the first name entered had a length of 15 bytes, then an additional 14 byte block would be added to the first. Thus, a 28 byte block of string space would be created when a 15 byte space is all that is really needed. This waste becomes even worse with numeric strings. Even if a one digit number is entered, a 13 byte string space is created.

When determining the size requirements of record fields with TMS9900 assembly language, you can and should make more efficient use of space. You can either define exact beginning and ending positions for fields, or where variable length fields are needed, you might use a special character to separate fields, or designate a byte which preceeds each field to contain the length of the field. For numeric items, if it is known that a value will only have a range of 1 to 255 for example, then the value can be stored as a binary expression in only one byte.

Any methodology that works for you can be valid, as long as you are consistent. Use your creativity and the flexibility of the TMS9900 language to design files which make efficient utilization of storage and memory. As long as the data files you create are to be used only by other TMS9900 assembly language programs which you have also coded, then any design that works for you will be acceptable. However, if you intend to process other kinds of files or feed the files you create to other kinds of programs, then you must know to what specification the file and record layout has been geared. When this information is not available, the dump utility from TI Programming Aids II can be of great help. This utility is strongly recomended both as a handy tool and as a key to understanding how records and files are created by TI BASIC.

Relative record (random access) files allow direct access to any record on the file without the need to read through the entire file as is the case with sequential files. Not only can this feature be used to randomly access a single record, but it can also be used to position the read/write head of the disk drive on a particular record. Then, you could read the following records sequentially. Relative record files can also be read from front to back, or back to front sequentially as well. All relative record files must utilize fixed length records. PAB bytes 4,5 (record length and character count) should be equal.

The coding and establishment of PABs and buffers for relative record files is identical to that of the previous example. The design of a subroutine to perform the DSR can be exactly like that of the sequential example. The type of access you can perform on a relative file depends on how the file is opened. Input allows you to read only. Output allows you to write only. Update lets you read and write. Append will let you add records to the end of a previously created file.

The key to random access is manipulation of the relative record number in PAB bytes 6 and 7. If these bytes contain all zeroes on your first read, and you issue successive reads without altering the relative record number, DSRLNK will increment the value for you. To know which record you are about to read or write, get the relative record number from PAB bytes 6,7 before each DSRLNK. Here are some instructions which do this.

```
RELREC  BSS 2              STORAGE FOR NUMBER
        •
        •
        LI    RØ,PABADR+6  ADDRESS IN PAB OF
                             RELATIVE RECORD NUMBER
        LI    R1,RELREC
        LI    R2,2         LENGTH OF READ - 2 BYTES
        BLWP @VMBR         GET THE NUMBER
```

Likewise, if you wish to read a specific record from a relative file, you must place the relative record number into PAB bytes 6,7 before the DSRLNK is executed. The above instructions will work for this if you replace VMBR with VMBW. The effect of trying to read a non-existent relative record would be an end of file error (5) and DSRLNK will close the file.

CASSETTE DSR

The DSR routine for cassettes is located in GROM and is one of the resident GPL (Graphics Programming Language) routines. When creating files on cassette several restrictions must be observed. Cassette files must be of fixed record length and the record length must be a multiple of 64 (64, 128 or 192). Cassette files can only be opened as input or output. There is no end of file detection with cassettes. You must create an end of file record as the last record on a cassette file and code any program which reads the file to detect whatever end of file data you created.

To access the cassette DSR or any of the other GPL routines, include a REF GPLLNK in your program when using the Editor/Assembler or EQUate the address >6018 to some valid two character label with the Line-by-Line assembler. When using the cassette DSR or other GPL routines, the automatic run feature (including the entry point address with the END directive) cannot be used.

The biggest difference with cassette file handling is the cassette DSR itself. Remember when coding the PAB data for your cassette files to use the correct value for PAB byte 8 (screen offset). For a stand-alone assembly program, this value should be >00. If your assembly program is called from BASIC, then this value must be >60. Here is a program example which uses a cassette file. This program writes 10 records to a cassette device. In truth, this program does not create any valid data for each record. It is merely an example of cassette access. Any actual data gathering or manipulation is up to you.

```
           REF   GPLLNK,VSBW,VMBW,KSCAN
   STATUS  EQU   >837C
   FAC     EQU   >834A
   PABBUF  EQU   >1ØØØ
   PAB     EQU   >F8Ø
   PDATA   DATA  >ØØØ2,>1ØØØ,>8Ø8Ø,>ØØØØ,>ØØØ3
   DEV     TEXT  'CS1          '
   RETURN  BSS   2
   WR      BSS   >2Ø
   START   MOV   R11,@RETURN
           LWPI  WR
```

```
12          LI   RØ,PAB      ESTABLISH PAB
13          LI   R1,PDATA
14          LI   R2,14
15          BLWP @VMBW
16          BL   @DSRCAS     OPEN FILE
17          LI   R1,>Ø3ØØ    ENABLE WRITE
18          LI   RØ,PAB
19          BLWP @VSBW
2Ø          CLR  R6
21 CPUT     BL   @DSRCAS     WRITE A RECORD
22          INC  R6          ADD 1 TO RECORD COUNT
23          CI   R6,1Ø       CHECK FOR 1Ø RECORDS WRITTEN
24          JLE  CPUT
25          LI   R1,>Ø1ØØ    ENABLE CLOSE
26          LI   RØ,PAB
27          BLWP @VSBW
28          BL   @DSRCAS     CLOSE FILE
29          JMP  EOJ         GO TO END OF JOB
3Ø DSRCAS   CLR  @STATUS     CLEAR GPL STATUS BYTE
31          CLR  @>83DØ      CLEAR ADDRESS >83DØ
32          LI   R3,3
33          MOV  R3,@>8354   PLACE VALUE OF 3 AT >8354
34          MOV  @DEV,@FAC     PLACE DEVICE NAME AT FAC
35          MOVB @DEV+2,@FAC+2
36          LI   R3,>Ø8ØØ
37          MOVB R3,@>836D   PLACE VALUE OF 8 AT BYTE >836D
38          LI   R3,PAB+13
39          MOV  R3,@>8356   ESTABLISH DSR POINTER
4Ø          BLWP @GPLLNK     PERFORM CASSETTE DSR
41          DATA >ØØ3D
42 *
43          RT               RETURN
44 EOJ      CLR  @STATUS     CLEAR STATUS
45          BLWP @KSCAN      WAIT FOR SOME KEY TO BE PRESSED
46          MOV  @STATUS,@STATUS
47          JEQ  EOJ
48          CLR  @STATUS
49          MOV @RETURN,11
5Ø          RT               RETURN
51          END
```

The heart of this program example is the cassette DSR routine at lines 30 through 43. The GPL status byte and address >83D0 must be all zeroes. The device name ('CS1") must be placed at address >834A and the length of the device name (3) must be at >8354, >8355. The value 8 must be in the byte at address >836D to indicate a DSR call. The PAB pointer address must be placed at address >8356 as in the other DSR examples. However, with the cassette DSR, this value must point to the byte after the device name "CS1" (PAB + 13). GPLLNK is invoked with a BLWP instruction and needs the value >3D passed to it to complete the instruction.

In this program example, the open causes the "REWIND CASSETTE" prompt to appear. The write prompts "PRESS CASSETTE RECORD," and the close prompts "PRESS CASSETTE STOP."

EDITOR/ASSEMBLER MANUAL REFERENCES

The following references will provide you with some more information on file handling.

Section 16.2.2 page 251
Section 16.2.4 page 262
Section 16.5 page 270 through Section 16.5.4 page 271

Notes on GROM access:

1. Accesses to GROM/GRAM must be separated by at least one instruction to accomodate differences in hardware performance.

2. GROM addressing is auto-incrementing. That is, after each access to a GROM address the address is automatically incremented by one by the computer.

3. GROM addresses are written most significant byte first.

4. When the GROM/GRAM read address (GRMRA) is read, the GROM address is destroyed and must be restored if required by the program.

Section 18.1 page 291 through Section 18.3 page 303
Section 24.12 page 443 through Section 24.12.4 page 444

Look up these terms in the glossary:

Device Service Routine
DSR
Field
File
GPL
GROM
Mode Of Operation
Peripheral Access Block
PAB

CHAPTER TEN SORTING AND HANDLING ARRAYS

The sequencing of data into a prescribed order, or sorting, is a prime example of table handling. A recurring theme throughout this text is the edge which TMS9900 assembly language holds over TI BASIC in speed and performance. Perhaps no other task can better illustrate the speed of TMS9900 assembly language than sorting. Conversely, sorting is one of the best examples of the shortcomings of TI BASIC in terms of computing speed.

Sorting is a task which involves a lot of computer overhead, a large number of repetitive actions which are time consuming in any language. There are several approaches to sorting which seek to alleviate the number of repetitions required to accomplish the sort. None of these various approaches involve any magical gimmicks and they all rely to a large extent on the processor involved to rapidly process the logical actions involved. As the number of records involved grows so does the amount of time required to sort them. Since speed and performance are critical to sort performance, the logical choice of a language to write the sort program in is assembly.

It is not the intent of this section to teach you all about sort theories but rather to illustrate how a simple sort can be implemented by using TMS9900 assembly language and the various instructions, subroutines, and methodologies covered so far. If you want to get serious about writing sort programs and learn some of the various approaches to sorting which are available, there are many books devoted to sorting, merging, and other processing of lists. Some of these are:

Lorin, Harold, SORTING AND SORT SYSTEMS, Reading, MA: Addison-Wesley, 1973.

Knuth, D. E., THE ART OF COMPUTER PROGRAMMING, Reading, MA: Addison-Wesley, 1973.

Flores, Ivan, COMPUTER SORTING, Englewood Cliffs, NJ: Prentice-Hall, 1969.

The sort program given in this section does not use any instructions or routines which have not already been covered. Some of the routines whose names you will recognize have been enhanced over previous examples or are used in new and different ways. Things such as program titles, prompts, color graphics, keyboard interaction, and file handling are all based on the examples and explanations given in previous sections. The actual sort logic used is rather crude compared to some of the more advanced techniques which are available. This simplistic approach provides a more easily understandable model and still runs quite fast because of the inherent speed of the TMS9900 assembly language and the TMS9900 microprocessor.

The file of data to be sorted is very similar to the first and last name file example of a previous section. However, to facilitate faster processing it is assumed to be a relative file with fixed length 80 byte records. In larger computers with much higher storage capacities it is feasible to process most any format file by loading its data into some work area in the form of a table and manipulating the data from there on. With the small number of records on the example file this approach could be taken using the home computer as well. The relative file format allows for the processing of a file as if it were already a table and uses the storage device (disk) for storage of the table rather than CPU RAM.

One of the basic tricks of sorting which eludes many beginners is that you do not need to sort entire records of data. The only data needing to be sorted is that data which makes up the sort key, the field or fields on which the file is to be sorted. This is providing of course that each of these sort keys can be associated with the record to which it belongs. This is where the relative record number (which may be thought of as a table subscript) comes into play. The basic premise of this program is to create an internal table, each entry of which is comprised of the sort key and relative record number of each record. This data is then sorted into the desired sequence. The table is then processed sequentially to get each record by way of its relative record number, and a new file is written which is in the sorted sequence. The storage capacity of a single-sided single-density diskette is about 90K bytes. This amount is far greater than the available CPU storage, so it is impossible to load this many bytes worth of records into memory at once. With this type of sort program however, tackling a file this large is feasible. In an assembly program, a record number stored in binary would occupy one word or two bytes. If the diskette contains 358 records of 250 bytes each, and the sort key data for each record is 20 bytes in length, then the amount of storage needed for the sort table would be (Sort Key Length + Relative Record Length) x Number of Records, or, (20 + 2) x 358 = 7,876 bytes.

Some sort approaches allow for in-place sorting of the original file. This program creates instead a new sorted file and preserves the original or "master" file's integrity. One drawback to the in-place sort is that, if for some reason, a processing error occurs, the original file might be damaged or lost if no backup copy was made. This approach also allows for the original data file to exist as many different files, each in its own sequence. This can be useful when it is necessary to process the same data from more than one sort key.

Here is the sort program.

```
         DEF   SORT
         REF   VMBW,VWTR,VMBR,VSBW,KSCAN,DSRLNK,VSBR
KEYVAL   EQU   >8375
STATUS   EQU   >837C
PABIN    EQU   >F80
PABOUT   EQU   >FA0
RECBUF   EQU   >1000
PNTR     EQU   >8356
CLOSB    BYTE  >01
WRITB    BYTE  >03
READB    BYTE  >02
ENTV     BYTE  >0D
LEFTV    BYTE  >08
RITEV    BYTE  >09
CURVAL   BYTE  >1E
REDOV    BYTE  >06
QUITV    BYTE  >05
         EVEN
CURPAT   DATA  >007E,>4242,>4242,>7E00
BORDER   DATA  >FFFF,>2020,>2020,>2020
         DATA  >2020,>2020,>2020,>2020
         DATA  >2020,>2020,>2020,>2020
         DATA  >2020,>2020,>2020,>FFFF
ERRMSG   TEXT  ' I/O ERROR ! CODE=    '
TIT1     TEXT  ' Chapter 10 Sort Program    '
TIT2     TEXT  ' Output File/Device Name?   '
TIT3     TEXT  '       Sorting Now          '
TIT4     TEXT  '        End Of Job          '
TIT5     TEXT  '  Press Any Key To Continue '
TOT1     TEXT  '       Record Count         '
TOT2     TEXT  '======Input===Output========'
ORD1     TEXT  ' <A>scend    <D>escend  >   '
EQLN     TEXT  '============================'
WATFLD   TEXT  ' What Field# To Sort On >   '
FIELD1   TEXT  ' First name ------------1   '
FIELD2   TEXT  ' Last  name ------------2   '
VERIFY   TEXT  ' Screen Complete - Redo?    '
EOF      DATA  >0000
SAVRTN   DATA  >0000
FLDLEN   DATA  >000E
TBFEND   DATA  >000D
TABEND   DATA  >000F
TABLEN   DATA  >0010
BLINK    DATA  >0BA0
DTEN     DATA  >000A
RELREC   DATA  >0000
TBMARK   DATA  >FFFF
INFILE   DATA  >0005,RECBUF,>5050,>0000,>000A
```

```
        TEXT  'DSK1.FILE1                  '
OUTFLE  DATA  >0003,RECBUF,>5050,>0000,>0000
        TEXT  'DSK                         '
OPTION  BSS   2
INBUFF  BSS   80
FNAME   EQU   INBUFF
LNAME   EQU   INBUFF+14
OUTBUF  BSS   80
HLDTAB  BSS   >10
MYREG   BSS   >20
SRTTAB  BSS   >1000
SORT    MOV   R11,@SAVRTN     SAVE RETURN ADDRESS
        LWPI  MYREG           LOAD WORKSPACE POINTER
        LI    R0,>0766        SET BORDER COLOR TO DARK RED
        BLWP  @VWTR
        LI    R0,>0380        SET CHAR SETS TO WHITE/BLUE
        LI    R1,>F400
LOOP1   BLWP  @VSBW
        INC   R0
        CI    R0,>039F        SET BORDER CHARACTER TO RED
        JLT   LOOP1
        LI    R1,>6600
        BLWP  @VSBW
        LI    R0,>08F0        LOAD CURSOR PATTERN
        LI    R1,CURPAT
        LI    R2,8
        BLWP  @VMBW
PROMPT  BL    @SCREEN         SET UP DISPLAY GRAPHICS
        LI    R0,66
        LI    R1,TIT1
        BLWP  @VMBW           DISPLAY PROGRAM TITLE
        LI    R0,130
        LI    R1,TIT2         PROMPT FOR OUTPUT DEVICE
        BLWP  @VMBW
        LI    R0,195
        LI    R10,15
        BL    @CURSOR         GET OUTPUT DEVICE NAME
        MOV   R7,@OUTFLE+8    MOVE FILE DESC LGH TO PAB
        LI    R0,195
        LI    R1,OUTFLE+10
        MOV   R7,R2
        BLWP  @VMBR           READ DEVICE NAME INTO PAB
WHTFLD  LI    R0,258          WHICH FIELD TO SORT ON?
        LI    R1,WATFLD
        LI    R2,28
        BLWP  @VMBW
        LI    R0,290
        LI    R1,FIELD1
        BLWP  @VMBW
        LI    R0,322
        LI    R1,FIELD2
        BLWP  @VMBW
        LI    R10,1
        LI    R0,282
        BL    @CURSOR        GET FIELD # CHOICE
        CI    R9,>3100       VERIFY RESPONSE = "1" OR "2"
        JLT   WHTFLD
        CI    R9,>3200
        JGT   WHTFLD         IF NOT, PROMPT AGAIN
ORDER   LI    R0,418         PROMPT ASCEND/DESCEND ORDER?
        LI    R1,ORD1
        BLWP  @VMBW
        LI    R10,1
        LI    R0,443
        BL    @CURSOR        GET ORDER CHOICE
        CI    R9,>4100       VERIFY RESPONSE = "A" OR "D"
        JLT   ORDER
        JEQ   VERIF
        CI    R9,>4400
        JNE   ORDER          IF NOT, PROMPT AGAIN
VERIF   LI    R0,706
        LI    R1,VERIFY
        BLWP  @VMBW          VERIFY CHOICES MADE
        CLR   R10
        BL    @CURSOR
        LI    R0,443
        BLWP  @VSBR
        MOVB  R1,@OPTION     SAVE ORDER OPTION
        LI    R0,282
        BLWP  @VSBR
        MOVB  R1,@OPTION+1   SAVE FIELD OPTION
        LI    R0,PABIN       ESTABLISH INPUT FILE PAB
        LI    R1,INFILE
        LI    R2,20
        BLWP  @VMBW
        BL    @DSRIN         OPEN INPUT
        MOVB  @READB,R1
        BLWP  @VSBW          PUT READ OP-CODE INTO PAB+0
        BL    @SCREEN        REDO SCREEN GRAPHICS
        LI    R0,226
        LI    R1,TOT1        DISPLAY RECORD COUNT HEADING
        BLWP  @VMBW
        LI    R0,258
        LI    R1,TOT2
        BLWP  @VMBW
        LI    R8,SRTTAB      DEST. ADDRESS FOR MOVWRD RTN
        MOV   @FLDLEN,R9     FIELD LGTH FOR MOVWRD RTN
        CLR   @EOF
READ    LI    R0,PABIN+6     PAB ADDRESS OF REL RECORD #
        LI    R1,RELREC
        LI    R2,2
        BLWP  @VMBR          GET RELATIVE RECORD NUMBER
        MOV   @RELREC,R4
        LI    R3,295
        LI    R0,300
        BL    @FIGUR         DISPLAY INPUT RECORD COUNT
        BL    @DSRIN         READ A RECORD
        MOV   @EOF,@EOF      CHECK FOR END OF FILE
        JNE   DONE1          IF EOF GO TO "DONE1"
        LI    R0,RECBUF      VDP RAM BUFFER ADDRESS
        LI    R1,INBUFF      CPU RAM BUFFER ADDRESS
        LI    R2,80          RECORD LENGTH
        BLWP  @VMBR          GET RECORD
        LI    R1,>3100       CHECK FOR WHICH FIELD
        CB    R1,@OPTION+1
        JNE   MOVLNM
        LI    R7,FNAME       FIRST NME ADDR. FOR MOVWRD
        JMP   MOVS
MOVLNM  LI    R7,LNAME       LAST NME ADDR. FOR MOVWRD
MOVS    BL    @MOVWRD        DO MOVWRD - LOAD SORT TABLE
        MOV   @RELREC,*R8+   LOAD REL RECORD # INTO TABLE
        JMP   READ           READ ANOTHER RECORD
DONE1   MOV   @TBMARK,*R8    LOAD END OF TABLE MARKER
        LI    R0,322
        LI    R1,EQLN
        LI    R2,28
        BLWP  @VMBW          DISPLAY "SORTING NOW"
        LI    R0,354
        LI    R1,TIT3
        BLWP  @VMBW
        LI    R5,>4400       LOAD R5 WITH VALUE OF "D"
COMPAR  LI    R1,SRTTAB      FIRST TABLE ENTRY ADDRESS
GETTAB  MOV   R1,R3          MOVE ADDR. VALUE IN R1 TO R3
        MOV   R1,R2          MOVE ADDR. VALUE IN R1 TO R2
        A     @TBFEND,R2     CALC. TABLE ENTRY LENGTH
        MOV   R3,R4
        A     @TABLEN,R4     CALC. NEXT TABLE ENTRY ADDR.
        C     @TBMARK,*R4    CHECK FOR END OF TABLE
        JEQ   DONE2          IF END OF TABLE, GO TO DONE2
        MOV   R3,R0          SAVE "A" ADDRESS IN R0
        MOV   R4,R1          SAVE "B" ADDRESS IN R1
ORDCHK  CB    R5,@OPTION     CHECK FOR ASCEND/DESCEND
        JEQ   CLOPD
*   ASCENDING ORDER   *
CLOP    C     R3,R2          CHECK FOR LAST BYTE OF FIELD
        JEQ   GETTAB         IF =, GET NEXT TABLE ENTRY
        CB    *R3+,*R4+      COMPARE A BYTE OF "A" TO "B"
        JEQ   CLOP           IF EQUAL, REPEAT
        JGT   SWIT           IF A > B, SWITCH
        JMP   GETTAB         IF A < B THEN GET NEXT ENTRY
```

```
  *   DESCENDING ORDER  *
CLOPD   C     R3,R2          CHECK FOR LAST BYTE OF FIELD
        JEQ   GETTAB         IF EQUAL, GET NEXT ENTRY
        CB    *R3+,*R4+      COMPARE A BYTE OF "A" TO "B"
        JEQ   CLOPD          IF EQUAL, REPEAT
        JGT   GETTAB         IF A > B THEN GET NEXT ENTRY
SWIT    MOV   R1,R7          LOAD R7 WITH "B" ADDRESS
        LI    R8,HLDTAB      LOAD R8 WITH HLDTAB ADDRESS
        MOV   @TABLEN,R9     LOAD R9 WITH LENGTH OF MOVE
        BL    @MOVWRD        MOVE "B" TO HOLD AREA
        MOV   RØ,R7          LOAD "A" ADDRESS
        MOV   R1,R8          LOAD "B" ADDRESS
        BL    @MOVWRD        MOVE "A" TO "B"
        LI    R7,HLDTAB      LOAD HOLD ADDRESS
        MOV   RØ,R8          LOAD "A" ADDRESS
        BL    @MOVWRD        MOVE HOLD (B) TO "A"
        JMP   COMPAR         GO BACK AND CHECK TABLE SEQ
MOVWRD  MOV   R9,R6          LOAD MOVE LENGTH
        A     R7,R6          CALC. MAXIMUM ADDRESS
MOVEM   MOV   *R7+,*R8+      MOVE A WORD & INC ADDRESSES
        C     R7,R6          CHECK FOR MAXIMUM ADDRESS
        JNE   MOVEM          IF NOT MAX, MOVE AGAIN
        RT                   ELSE, RETURN
DONE2   CLR   R9             CLEAR REGISTER 9
        LI    RØ,PABOUT      ESTABLISH OUTPUT FILE PAB
        LI    R1,OUTFLE
        LI    R2,25
        BLWP  @VMBW
        BL    @DSROUT        OPEN OUTPUT FILE
        MOVB  @WRITB,R1      PUT WRITE OP-CODE TO PAB+Ø
        BLWP  @VSBW
        LI    RØ,PABIN       RE-ESTABLISH INPUT FILE PAB
        LI    R1,INFILE
        BLWP  @VMBW
        BL    @DSRIN         OPEN INPUT FILE
        MOVB  @READB,R1      PUT READ OP-CODE TO PAB+Ø
        BLWP  @VSBW
        LI    R8,SRTTAB+14   TABLE ADDR. OF REL RECORD #
GETRR   MOV   R8,R4          MOVE ADDRESS IN R8 TO R4
        S     @FLDLEN,R4     CALC. ADDR. OF FIRST ENTRY
        C     @TBMARK,*R4    CHECK FOR END OF TABLE
        JEQ   EOJ            IF END OF TABLE, GO TO EOJ
        MOV   R8,R1          MOVE REL RECORD # ADDR. TO R1
        LI    RØ,PABIN+6     LOAD RØ WITH DEST. ADDR.
        LI    R2,2           LOAD R2 WITH LENGTH OF DATA
        BLWP  @VMBW          PUT REL RECORD # INTO PAB+6
        BL    @DSRIN         READ THAT RECORD
        INC   R9                 INCREMENT OUTPUT REC COUNT
        BL    @DSROUT            WRITE OUTPUT RECORD
        A     @TABLEN,R8         CALC. NEXT REL REC. ADDR.
        MOV   R9,R4              MOVE RECORD COUNT TO R4
        LI    R3,304
        LI    R0,309
        BL    @FIGUR             DISPLAY OUTPUT RECORD COUNT
        JMP   GETRR              GET NEXT REL RECORD NUMBER
EOJ     LI    R0,354             DISPLAY "END OF JOB"
        LI    R1,TIT4
        LI    R2,28
        BLWP  @VMBW
        LI    R0,706
        LI    R1,TIT5
        BLWP  @VMBW
        LI    R0,PABIN
        MOVB  @CLOSB,R1          PUT CLOSE OP-CODE TO PAB+0
        BLWP  @VSBW
        LI    R0,PABOUT          PUT CLOSE OP-CODE TO PAB+0
        BLWP  @VSBW
        BL    @DSRIN             CLOSE INPUT FILE
        BL    @DSROUT            CLOSE OUTPUT FILE
EOJX    CLR   R10
        BL    @CURSOR            PERFORM "PRESS ANY KEY"
EOJQ    CLR   @STATUS            CLEAR THE GPL STATUS BYTE
        MOV   @SAVRTN,R11        MOVE RETURN ADDRESS TO R11
XT      RT                       RETURN (B *R11)
CURSOR  CLR   R9                 CLEAR REGISTER 9
        CLR   @KEYVAL            CLEAR KEYSTROKE VALUE ADDR.
        CLR   @STATUS            CLEAR GPL STATUS BYTE
        MOV   R10,R10            CHECK R10 FOR ZERO VALUE
        JEQ   CURL1              IF ZERO, GO TO CURSOR LOOP1
        MOV   R0,R8              SAVE BEGINNING CURSOR ADDR.
        A     R8,R10             CALC. MAXIMUM CURSOR ADDR.
        MOV   R8,R7              MOVE BEGIN ADDR.TO ACCUM.
CURPUT  CLR   R6                 CLEAR REGISTER 6
        MOV   R7,R0              MOVE ACCUMULATOR TO R0
        LI    R1,>2000           LOAD REGISTER 1 WITH SPACE
        BLWP  @VSBW              DISPLAY SPACE
CURL1   BLWP  @KSCAN             PERFORM KEYBOARD SCAN
        MOVB  @STATUS,@STATUS CHECK FOR KEYSTROKE
        JNE   DETECT             IF KEY PRESSED, WHICH KEY?
        AI    R6,4               ADD 4 TO REGISTER 6
        C     R6,@BLINK          COMPARE R6 TO BLINK COUNT
        JLT   CURL1              IF LESS, REPEAT CURL1
        MOV   R10,R10            CHECK R10 FOR ZERO VALUE
        JEQ   CURL1              IF ZERO, GO TO CURL1
        CLR   R6                 CLEAR REGISTER 6
        MOVB  @CURVAL,R1       MOVE CURSOR CODE TO R1
        BLWP  @VSBW            DISPLAY CURSOR
CURL2   INC   R6               ADD 1 TO R6
        C     R6,@BLINK        COMPARE R6 TO BLINK COUNT
        JLT   CURL2            IF LESS, REPEAT CURL2
        JMP   CURPUT           REPEAT CURSOR LOOP
DETECT  CB    @REDOV,@KEYVAL   CHECK FOR "REDO" VALUE
        JEQ   REDOX            IF REDO, GO TO REDO EXIT
        CB    @QUITV,@KEYVAL   CHECK FOR "QUIT" VALUE
        JEQ   EOJQ             IF QUIT, GO TO EOJ/QUIT
        MOV   R1Ø,R1Ø          CHECK R1Ø FOR ZERO VALUE
        JEQ   XT               IF ZERO, RETURN
        CB    @ENTV,@KEYVAL    CHECK FOR "ENTER" VALUE
        JEQ   ENTER
        CB    @LEFTV,@KEYVAL   CHECK FOR "LEFT ARROW"
        JEQ   LEFT
        CB    @RITEV,@KEYVAL   CHECK FOR "RIGHT ARROW"
        JEQ   RITE
        C     R7,R1Ø           CHECK FOR MAX. CURSOR ADDR.
        JEQ   CURPUT           IF EQUAL, GO TO "CURPUT"
        MOV   R7,RØ            MOVE NEW ADDRESS TO RØ
        MOVB  @KEYVAL,R1       MOVE KEYSTROKE VALUE TO R1
        MOVB  @KEYVAL,R9       SAVE KEYSTROKE VALUE IN R9
        BLWP  @VSBW            DISPLAY KEYSTROKE CHARACTER
        INC   R7               ADD 1 TO ADDRESS ACCUMULATOR
        JMP   CURPUT           GO TO "CURPUT"
LEFT    C     R7,R8            CHECK FOR MIN CURSOR ADDR.
        JEQ   CURPUT           IF EQUAL, GO TO "CURPUT"
        MOV   R7,RØ            MOVE CURRENT ADDRESS TO RØ
        LI    R1,>2ØØØ         LOAD R1 WITH SPACE CHAR.CODE
        BLWP  @VSBW            WRITE A SPACE AT CURR. ADDR.
        DEC   R7               SUBTRACT 1 FROM CURR. ADDR.
        JMP   CURPUT           GO TO "CURPUT"
RITE    C     R7,R1Ø           CHECK FOR MAX CURSOR ADDR.
        JEQ   CURPUT           IF EQUAL, GO TO "CURPUT"
        INC   R7               ADD 1 TO CURRENT ADDRESS
        JMP   CURPUT           GO TO "CURPUT"
ENTER   LI    R1,>2ØØØ         LOAD R1 WITH SPACE CHAR CODE
        MOV   R7,RØ            MOVE CURRENT ADDRESS TO RØ
        BLWP  @VSBW            SPACE OVER CURSOR SYMBOL
        S     R8,R7            CALC. ACTUAL LENGTH OF DATA
        JEQ   CURSOR           IF ZERO, REPEAT CURSOR RTN
        RT                     ELSE, RETURN
REDOX   B     @PROMPT          BRANCH TO ADDRESS "PROMPT"
SCREEN  CLR   RØ               CLEAR REGISTER Ø
        LI    R1,BORDER        LOAD R1 WITH ADDR. OF BORDER
        LI    R2,32            LOAD R2 WITH LENGTH OF DATA
SCRL    BLWP  @VMBW            WRITE ONE LINE OF PATTERN
        CI    RØ,736           COMPARE RØ TO MAXIMUM VALUE
        JHE   LEAVE            IF = OR >, GO TO "LEAVE"
        AI    RØ,32        ADD 32 TO RØ
        JMP   SCRL         GO TO "SCRL"
LEAVE   LI    RØ,2         LOAD RØ WITH THE VALUE 2
        LI    R1,EQLN      LOAD R1 WITH GRAPHICS ADDR
        LI    R2,28        LOAD R2 WITH LENGTH OF LINE
        BLWP  @VMBW        DISPLAY AT SCREEN ADDRESS 2
        LI    RØ,738       LOAD RØ WITH THE VALUE 738
        BLWP  @VMBW        DISPLAY AT SCREEN ADDR 738
RTN     RT                 RETURN
DSRIN   LI    R6,PABIN+9   LOAD R6 WITH PAB ADDRESS
        JMP   DSR
DSROUT  LI    R6,PABOUT+9  LOAD R6 WITH PAB ADDRESS
DSR     MOV   R6,@PNTR     MOVE R6 TO POINTER ADDRESS
        BLWP  @DSRLNK      PERFORM DEVICE SERVICE RTN
        DATA  8
        JNE   RTN          IF NO ERRORS, RETURN
ERROR   MOV   R6,RØ        MOVE R6 TO RØ
        AI    RØ,-8        CALC. ADDRESS OF PAB BYTE 1
        BLWP  @VSBR        READ PAB BYTE 1 INTO R1
        SRL   R1,5         SHIFT R1 RIGHT 5 POSITIONS
        MOV   R1,@EOF      MOVE R1 TO THE EOF FLAG
        CB    @QUITV,R1    CHECK FOR EOF (VALUE >Ø5)
        JEQ   RTN          IF EQUAL, RETURN
        BL    @SCREEN      ELSE, REDO SCREEN GRAPHICS
        MOV   R6,RØ        MOVE R6 TO RØ
        INC   RØ           CALC. ADDRESS OF PAB BYTE 1Ø
        LI    R1,INBUFF    LOAD R1 - TEMP CPU RAM ADDR
        LI    R2,2Ø        LOAD R2 - LENGTH OF WRITE
        BLWP  @VMBR        GET FILE NAME IN ERROR
        LI    RØ,259       LOAD RØ - SCREEN ADDR 259
        BLWP  @VMBW        DISPLAY FILE NAME IN ERROR
        MOV   @EOF,R1      MOVE ERROR CODE TO R1
        AI    R1,>3ØØØ     MAKE IT AN ASCII NUMERAL
        LI    RØ,343       SCREEN ADDR FOR ERROR CODE
        BLWP  @VSBW        DISPLAY ERROR CODE
        LI    RØ,322       SCREEN ADDR FOR ERROR MSG
        LI    R1,ERRMSG    ADDRESS OF THE MESSAGE
        BLWP  @VMBW        DISPLAY ERROR MESSAGE
        B     @EOJX        GO TO EOJ
FIGUR   MOV   R4,R5        MOVE R4 TO R5
        CLR   R4           CLEAR R4
        DIV   @DTEN,R4     DIVIDE R4/R5 BY TEN
        AI    R5,>3Ø       MAKE R5 AN ASCII NUMERAL
        SLA   R5,8         SHIFT R5 LEFT 8 PLACES
        MOV   R5,R1        MOVE R5 TO R1
        BLWP  @VSBW        DISPLAY A DIGIT
        DEC   RØ           SUBTRACT 1 FROM RØ
        C     RØ,R3        COMPARE RØ TO R3
        JNE   FIGUR        IF NOT EQUAL REPEAT
        RT                 ELSE, RETURN
        END
```

At nearly 400 lines, this TMS9900 assembly program might appear formidable to the beginner. However, this program is not as complex as the number of lines would seem to indicate. In addition to performing sort logic, this program represents an example of everything that has been covered in this text so far. It includes a number of options and features besides sorting which add considerably to its length. In fact, the actual sorting instructions range from line 147 through 251, or only about 28 percent of the entire program. The rest of the program is comprised of the instructions, directives, and subroutines which have been previously detailed. Lastly, the key to understanding any program that you are looking at for the first time is to take it one line at a time. This is true whether the program is 5 or 5000 lines long. Do not be intimidated by a program's length. The only sections of this program to be detailed will be those which involve new logic (the sort) and new or enhanced applications of previously introduced instructions and subroutines.

Lines 1 through 59 use the various TMS9900 assembler directives to define the constants, addresses, data, and work areas this program will require. The data items include a new pattern description for the cursor, graphics characters for the screen display, PAB data for the input and output files, and the titles, prompts, and messages which will be displayed. Line 59 sets aside 4096 bytes (hex >1000) for the sort table space. This is an arbitrary value. When designing a sort program or any other program's internal workspace size, it is necessary to allow for some expected or maximum number of records and, therefore, bytes.

Lines 60 through 75 save the return address, load the workspace register pointer, set the foreground/background colors of the border characters and display characters, and load the cursor pattern data to the pattern descriptor table in VDP RAM. All these steps and instructions have been used before in previous program examples. The colors used in this program are white on blue for the displayable character set with a dark red screen border.

Line 76 performs a Branch and Link to the subroutine SCREEN. This routine will load the border pattern into the screen image table and write a line of "=" symbols across the top and bottom of the screen. This function is coded as a subroutine to allow the screen graphics to be established or redone at any point in the program without having to code the actual instructions more than once. Lines 79 through 90 display the program title and prompts the user for the file/device name for the sorted output file. While the file/device name of the input file is hard-coded into the program, the user is free to specify the output file. Since the sorted output file in this program example is a relative file like the input file, it naturally must be on disk. The file/device name is read from screen address 195 into the output file PAB byte 10. The length of this data is returned in R7 by CURSOR. This length value is moved to output file PAB bytes 8,9 to complete the output file PAB data.

In chapter eight you were introduced to a fundamental keyboard input routine called CURSOR, which provided interactive keyboard ability. This routine is again included in this program. This version of CURSOR includes some additional capabilities not found in the original. Besides using a unique symbol pattern for the cursor, the new CURSOR routine also makes the cursor blink. Two additional exits are allowed as well. If the user presses REDO the new CURSOR routine branches to line 76 (PROMPT) to let them re-enter the data which is required. Should the user press QUIT, the routine branches to line 268, EOJQ. Branching to this address bypasses the closing of the input and output files (which have not been opened yet) and ends the program. Lastly, so long as the response length specified in R10 before the BL to CURSOR is greater than zero, CURSOR will not allow the user to simply press ENTER without having entered some data.

Lines 91 through 107 ask the user to specify which field the file is to be sorted on. The prompts defined in lines 34, 35, and 36 allow the user to enter a "1" for first name, or a "2" for last name. The actual response is compared to the ASCII codes for these two numerals to verify that a valid choice has been made. If any value is entered other than those allowed, the prompt sequence is repeated. This has the same effect as pressing REDO at this point. The only other option available would be QUIT. Next, lines 108 through 118 ask that the user specify the order in which the records are to be sorted. A response of "A" indicates that the file is to be sorted in ascending sequence. A reponse of "D" means that the file is to be sorted in descending order. The actual responses are verified to be either "A" or "D". Up to this point, the program has required the user to tell it the file/device name of the sorted ouput file, which field to sort on, and whether the sort is to be in ascending or descending order.

Lines 119 through 123 ask that the user verify the information they have just entered, and respond by either pressing REDO or any other key. If the response is REDO, then CURSOR branches to the beginning of the prompt sequence to allow the data to be re-entered. If any other key is pressed the program continues. Lines 124 through 136 get and save the field and sequence options, establish the input file PAB in VDP RAM, open the input file, and enable the input file to be read.

Lines 137 through 144 get rid of the prompts and responses by repeating the screen graphics and display the headings for the record counts which will be taken and displayed during processing. A count will be taken of the number of input records (sort in) and the number of output records (sort out). These two counts should always be equal as long as the program is functioning properly and no processing errors occur.

The first phase of the sort process involves reading the file to be sorted and building a table from the sort field and relative record number for each record. Because many repetitious word (16 bit) moves are involved in loading the table and performing the actual sort, a subroutine called MOVWRD has been coded just for this purpose. Whenever the same or similar set of instructions needs to be executed more than once, it is preferable to create a subroutine comprised of the instruction set rather than coding the instructions over and over again. The routine MOVWRD uses registers 6, 7, 8 and 9 to move words of memory between various CPU RAM locations. Register 7 must contain the origin address, register 8 the destination address, register 9 the length of the move in bytes (always an even number for word moves), and register 6 is used for calculations.

Line 144 loads R8 with the destination address (sort table area, SRTTAB). Line 145 places the length of the data into R9. Lines 162 through 167 put the origin address into R7 depending on which field has been selected to be sorted on. The length of the sort field is set to 14 (>E) for either field. This is the maximum length of the first name field as defined in the TI BASIC program which created it. The actual length of the last name field is variable. The length of the sort field determines the number of bytes which must be compared to detect those records which are out of sequence. The larger this number, the more time the sort will take. One shortcut which can be taken in setting up sort logic is to only sort on a given number of bytes regardless of the length of the sort field. Since it is unlikely that any record will contain a last name greater than 14 bytes, and even more unlikely that two last names would contain the same characters until the fifteenth position, this approach is reasonable, accurate, and time saving. The length value could be less or greater, but too much of a deviance could be inaccurate or time consuming. The actual number of bytes you choose to compare is a matter of judgement, but it is possible to cheat a little.

To demonstrate how the sort logic in this program operates, it is necessary to make a few assumptions about which options have been selected, and provide some data for the program to process. For the rest of this explananation, assume that the user wants to sort the input file on the last name field, and wants the file sorted in ascending order. Here is an input file of 10 records.

Relative Record #	First Name	Last Name
0000	JIM	SMITH
0001	BOB	JONES
0002	MARY	QUEENS
0003	MARVIN	STONE
0004	LINDA	BLACKSMITH
0005	GEORGE	WASHINGTON
0006	THOMAS	JEFFERSON
0007	WILLIAM	MCKINLEY
0008	HOWARD	TAFT
0009	HOWARD	JOHNSON

As you can see, the records are not in any particular order. Lines 147 through 170 read the file and build the sort table. Each table entry is composed of the first 14 bytes of the last name field plus 2 bytes for the relative record number of each record for a total length of 16 bytes for each table entry. When the end of the file is reached, line 171 moves a word containing >FFFF to the next table address in order to mark the end of the table. The subroutine MOVWRD, which is used to create the table, illustrates the use of an addressing mode known as Workspace Register Indirect Auto-Increment. The line: 0216 MOVEM *R7+,*R8+ specifies that the values in each register are to be used as addresses and that after each move the values in the registers will be automatically incremented. This handy addressing feature increments the registers involved by 1 for byte instructions and by 2 for word instructions. After the file has been read and the sort table is created, the sort table would appear as below. Note that the actual values in the sort table are ASCII codes for the letters and binary expressions for the relative record numbers. The example here is shown in display format.

```
ADDRESS       TABLE CONTENTS

              ________________
             |                |
SRTTAB+0     |SMITH         00|
             |----------------|
SRTTAB+16    |JONES         01|
             |----------------|
SRTTAB+32    |QUEENS        02|
             |----------------|
SRTTAB+48    |STONE         03|
             |----------------|
SRTTAB+64    |BLACKSMITH    04|
             |----------------|
SRTTAB+80    |WASHINGTON    05|
             |----------------|
SRTTAB+96    |JEFFERSON     06|
             |----------------|
SRTTAB+112   |MCKINLEY      07|
             |----------------|
SRTTAB+128   |TAFT          08|
             |----------------|
SRTTAB+144   |JOHNSON       09|
             |----------------|
SRTTAB+160   |>FFFF           |
             |________________|
```

The second phase of this sort process is to arrange the table entries into the desired order. To do this, the program starts with the first table entry and compares each byte of the sort key data to that of the next table entry. If the sort key data of the two table positions being compared is already in the correct sequence, or if the two are equal, then no action is required and the comparison moves on to the next pair of table entries. To make it easier to keep the relationship of the entries being compared straight, the first table entry will be referred to as "A" and the next table entry as "B". For the ascending sequence example, whenever A is found to be greater than B, their positions are switched in the table.

Lines 180 through 185 calculate the table addresses of the pair of table entries to be compared. The "A" address is saved in R0, and the "B" address is saved in R1. Lines 186,187 test for the end of the table marker. If the end of the table has been reached then this phase of the sort is done, and line 187 directs the program to address "DONE2". The first time through this logic, R3 is loaded with address SRTTAB, and R4 is loaded with address SRTTAB + 16. R2 will contain the address value of the last byte of the sort key portion of the table entry. If R2 is equal to R3 at line 192, then all the bytes of the sort key have been compared and it is time to move on to the next pair of table entries. Lines 190 and 191 check the order option again by comparing the option to >44 (ASCII for "D"). If descending order was selected, then the compare loop CLOPD is performed. Otherwise, the default is ascending order, which performs compare loop CLOP. Here is a diagram of the first comparison.

```
Ø194       CB  *R3+,*R4+
Ø195       JEQ CLOP
Ø196       JGT SWIT
```

ADDRESS IN REGISTER 3	VALUE AT ADDRESS	ADDRESS IN REGISTER 4	VALUE AT ADDRESS
SRTTAB+Ø	>53 or "S"	SRTTAB+16	>4A or "J"

The Compare Bytes instruction is used because the comparison must be done on a character by character basis, and one character takes one byte. The value at the address in R3 ("S") is greater than the value at the address in R4 ("J"). This comparison results in the greater-than bit being set in the status register. This condition is checked and the resulting action is directed by line 196. It will be necessary to exchange the positions of the two table entries. Using the A and B naming convention, the table entry for SMITH is "A", and JONES is "B". There are three steps to the switch process:

Move "B" to a hold area - HLDTAB.

```
     |_______________|
"A"  |SMITH        ØØ|
     |_______________|

     |_______________|                              |_______________|
"B"  |JONES        Ø1|---------->"HLDTAB"           |JONES        Ø1|
     |_______________|                              |_______________|
```

Move "A" to "B".

```
     |_______________|
"A"  |SMITH        ØØ|
     |_______________|
             |
             |
             V
     |_______________|                              |_______________|
"B"  |SMITH        ØØ|            "HLDTAB"          |JONES        Ø1|
     |_______________|                              |_______________|
```

Move TABHLD ("B") to "A"

```
     _______________                 _______________
    |               |               |               |
"A" |JONES        Ø1|<----------"HLDTAB" |JONES        Ø1|
    |_______________|               |_______________|

     _______________
    |               |
"B" |SMITH        ØØ|
    |_______________|
```

These are the instructions which perform the switch logic.

```
Ø2Ø3 SWIT    MOV   R1,R7          LOAD R7 WITH "B" ADDRESS
Ø2Ø4         LI    R8,HLDTAB      LOAD R8 WITH HLDTAB ADDRESS
Ø2Ø5         MOV   @TABLEN,R9     LOAD R9 WITH LENGTH OF MOVE
Ø2Ø6         BL    @MOVWRD        MOVE "B" TO HOLD AREA
Ø2Ø7         MOV   RØ,R7          LOAD "A" ADDRESS
Ø2Ø8         MOV   R1,R8          LOAD "B" ADDRESS
Ø2Ø9         BL    @MOVWRD        MOVE "A" TO "B"
Ø21Ø         LI    R7,HLDTAB      LOAD HOLD ADDRESS
Ø211         MOV   RØ,R8          LOAD "A" ADDRESS
Ø212         BL    @MOVWRD        MOVE HOLD (B) TO "A"
Ø213         JMP   COMPAR         CHECK TABLE SEQUENCE AGAIN
Ø214 MOVWRD  MOV   R9,R6          LOAD MOVE LENGTH
Ø215         A     R7,R6          CALC. MAXIMUM ADDRESS
Ø216 MOVEM   MOV   *R7+,*R8+      MOVE WORD,INCREMENT ADDRESS
Ø217         C     R7,R6          CHECK FOR MAXIMUM ADDRESS
Ø218         JNE   MOVEM          IF NOT MAX, MOVE AGAIN
Ø219         RT                   ELSE, RETURN
```

Line 213 returns to the label COMPAR after the switch has been done. Each time a pair of table entries are found to be out of order and are switched, the comparison logic starts all over again at the beginning of the table. This is done over and over again until all the table entries are in the correct order. If all the entries are in correct order there is never a condition which causes the switch logic to execute and the end of the table is reached. The type of sort algorithm being applied to the sort table is known as a "Bubble" sort. Each item in the table is moved up to the position where it belongs. You may come across many different programs which are all called bubble sorts. Their specifics will vary, but the general approach is the same. When this second phase of the sort program is completed, the sort table now looks like this.

```
              |                |
SRTTAB+Ø      |BLACKSMITH    Ø4|
              |----------------|
SRTTAB+16     |JEFFERSON     Ø6|
              |----------------|
SRTTAB+32     |JOHNSON       Ø9|
              |----------------|
SRTTAB+48     |JONES         Ø1|
              |----------------|
SRTTAB+64     |MCKINLEY      Ø7|
              |----------------|
SRTTAB+8Ø     |QUEENS        Ø2|
              |----------------|
SRTTAB+96     |SMITH         ØØ|
              |----------------|
SRTTAB+112    |STONE         Ø3|
              |----------------|
SRTTAB+128    |TAFT          Ø8|
              |----------------|
SRTTAB+144    |WASHINGTON    Ø5|
              |----------------|
SRTTAB+16Ø    |>FFFF           |
              |________________|
```

The 3rd phase involves reading the table sequentially, and using the relative record numbers to randomly retrieve records from the input file and write them out to the output file.

Lines 220 through 233 prepare for phase III by establishing the input and output file PABs, opening the input and output files, and setting the input file op-code to Read and the output file op-code to Write. In phase I the input file was read until end of file and was closed by the DSR routine. Now it is to be read again, so the PAB data is re-established and the file opened.

Line 234 loads the beginning table address for the first relative record number entry into R8. Lines 235 through 238 check for the end of table marker each time through the loop GETRR. Lines 239 through 243 get the relative record number from the table, place it into PAB bytes 6,7 of the input file, and read that record from the file. The PAB data for the input file names the VDP RAM area RECBUF as the buffer space into which each read operation places a record from the input file. The PAB data for the output file uses the same buffer area as the place where each write operation expects to find the record to be written. No other manipulation of the record data is required. Here is a diagram of the first GETRR loop.

Get (read) the record number found at the address in R8 from the input file into RECBUF.

```
 REGISTER 8              SRTTAB          INPUT  FILE
 ----------              ------
                        |----|          |______________________________|
  SRTTAB+14         +14 | Ø4 |          |ØØ JIM          SMITH          |
                        |----|          |------------------------------|
                    +3Ø | Ø6 |          |Ø1 BOB          JONES          |
                        |----|          |------------------------------|
                    +46 | Ø9 |          |Ø2 MARY         QUEENS         |
                        |____|          |------------------------------|
                                        |Ø3 MARVIN       STONE          |
                                        |------------------------------|
                         ____________   |Ø4 LINDA        BLACKSMITH     |
                        |               |------------------------------|
                        |               |Ø5 GEORGE       WASHINGTON     |
                        v               |______________________________|
              ________________________
             |                        |
RECBUF       |LINDA        BLACKSMITH |
             |________________________|
```

Write an output file record from RECBUF.

```
              ________________________
             |                        |
RECBUF       |LINDA        BLACKSMITH |
             |________________________|
                                         OUTPUT  FILE
                        |                ______________________________
                        |               |                              |
                        ----------->    |ØØ LINDA        BLACKSMITH     |
                                        |______________________________|
```

Add the value at TABLEN (16, hex >10) to R8 and repeat the loop.
REGISTER 8 now contains SRTTAB+30.

This process continues until the end of table marker is detected, at which time the program logic transfers to EOJ. The End of Job message is then displayed and the files closed. The message "Press Any Key To Continue" and a BL to CURSOR allows for a pause before actually ending the program. With the REDO capability of this version of CURSOR, the user may elect to repeat the entire sort program at this point or press any other key to end processing.

At the end of the program, the input file exists in its original state, and the new, sorted output file has been created.

```
        INPUT  FILE                               OUTPUT  FILE

Relative  First     Last               Relative  First      Last
Record #  Name      Name               Record #  Name       Name
-------------------------------        -------------------------------
ØØØØ      JIM       SMITH              ØØØØ      LINDA      BLACKSMITH
ØØØ1      BOB       JONES              ØØØ1      THOMAS     JEFFERSON
ØØØ2      MARY      QUEENS             ØØØ2      HOWARD     JOHNSON
ØØØ3      MARVIN    STONE              ØØØ3      BOB        JONES
ØØØ4      LINDA     BLACKSMITH         ØØØ4      WILLIAM    MCKINLEY
ØØØ5      GEORGE    WASHINGTON         ØØØ5      MARY       QUEENS
ØØØ6      THOMAS    JEFFERSON          ØØØ6      JIM        SMITH
ØØØ7      WILLIAM   MCKINLEY           ØØØ7      MARVIN     STONE
ØØØ8      HOWARD    TAFT               ØØØ8      HOWARD     TAFT
ØØØ9      HOWARD    JOHNSON            ØØØ9      GEORGE     WASHINGTON
```

The way in which the input/output requirements of the two files are addressed demonstrates a way in which Device Service Routines can be applied. In the section on file handling, it was stated that all DSR requests are handled the same for any device except cassettes and the the only differences were the particular file data (PAB) pointed to, and the operation (op-code) requested. In the above DSR routine, the only difference between the input and output file access are the lines 351,352,353, which set the pointer for the PAB (file description) used. The individual op-codes for read, write, and close were changed within the body of the program. The remaining instructions work for either file. Should an error be detected, the pointer address in R6 can be used to calculate the address of PAB byte 1 (the error code), and PAB byte 10 (the file description "DSK1.FILE1") If the output file was to have been written to a cassette device, this approach would not work. The cassette DSR must be accessed by a different set of instructions as outlined in the chapter on file handling.

The cursor subroutine as coded in this program example is an improved version of the one first introduced. This version operates much the same as the original. The most notable difference is the blinking cursor. Lines 279 through 288 write a space (hex >20) to the screen and count to 2,976 (hex >0BA0) by fours. Once the loop CURL1 has reached this value, lines 289 through 297 then display the cursor symbol. The loop CURL2 counts to 2,976 by ones. Some indication of the speed of TMS9900 assembly language is evidenced by the fact that all of this counting occurs in the space of a blink. Counting by fours while the space is displayed and then counting by ones while the cursor is displayed, means that 80 percent of the time the cursor is seen and 20 percent of the time the space is seen. You can alter the rate of blink by changing the value at BLINK, or by changing the incremental value for either CURL1 or CURL2. The keyboard scan utility is performed within the first loop so that the response to the pressing of a key is not too sluggish.

Lines 298 through 301 check for the QUIT and REDO values and direct program logic accordingly. When the actual length of the data entered is calculated at line 332, the equal bit will be set in the status register if the answer in R7 is zero. If an allowable value of zero was placed into R10 before the CURSOR subroutine logic was performed, then lines 329 through 334 are never reached. Therefore, if CURSOR reaches the label ENTER, a value greater than zero must have been specified. Line 333 checks for the value in R7, and if it is zero, the entire CURSOR subroutine is repeated.

CHAPTER ELEVEN
MIXING ASSEMBLY WITH BASIC

It is possible to create subprograms in assembly language that can be called from your TI BASIC or TI Extended BASIC programs. This can result in a hybrid variety of programs that offer the best of both worlds. Program tasks that can be accomplished with adequate efficiency in BASIC can be coded and debugged easily in that language. Tasks that require improved speed and performance, or tasks that simply are impossible to perform in BASIC, can be coded in assembly. The Editor/Assembler module and the Mini-Memory module both provide additional statements for use with TI BASIC. These statements facilitate linking assembly routines with BASIC programs. Likewise, TI Extended BASIC offers statements to facilitate this function that are not normally available in TI BASIC. When you are running an assembly program with Editor/Assembler or Mini-Memory, the computer's memory is not arranged exactly the same as when you are operating in TI BASIC or TI Extended BASIC. In order for your assembly language subprograms to function correctly in the BASIC environment, there are specific differences that you must be aware of.

First, you must determine which BASIC environment your program will be operating in. Of primary consideration to the assembly language programmer are the differences found in VDP RAM. The BASIC program and all its data, routines, and workspaces occupy a good portion of this area. Your assembly language program must not change the values of any addresses being used by BASIC. Certain VDP RAM value tables, such as the color table, are located at different addresses in BASIC than they are in stand-alone assembly programs.

Further, your assembly program must allow for screen bias and offset while running with BASIC. The value of this offset is always >60 or decimal 96. Any character your assembly language program wishes to display must have its character code value incremented by >60. Normally, to display an "A" on the screen, the character code >41 is used. In the BASIC environment, however, this value must be adjusted:

```
LI      RØ,293     LOAD RØ WITH VDP RAM SCREEN ADDRESS
LI      R1,>41ØØ   LOAD R1 WITH "A" CODE
AI      R1,>6ØØØ   ADJUST FOR BASIC
BLWP    @VSBW      DISPLAY THE "A"
```

For string constants such as those created with the TEXT directive, every byte of the string must be adjusted by adding >60 before it is written to the screen image table address.

VDP RAM UTILIZATION

ASSEMBLER MODE

ADDRESS HEX	ADDRESS DECIMAL	
>ØØØØ	Ø	SCREEN IMAGE TABLE
>Ø2FF	767	
>Ø3ØØ	768	SPRITE ATTRIBUTE
>Ø37F	895	
>Ø38Ø	896	COLOR TABLE AND FREE SPACE
>Ø3FF	1Ø23	
>Ø4ØØ	1Ø24	SPRITE DESCRIPTOR TABLE
>Ø7FF	1919	
>Ø8ØØ	192Ø	PATTERN DESCRIPTOR TABLE AND FREE SPACE
>ØFFF	4Ø95	
>1ØØØ	4Ø96	FREE SPACE PAB's AND BUFFERS
>137F	4991	
>138Ø	4992	PROGRAM FILE LOAD BUFFER
>34FF	13567	
>35ØØ	13568	DISK DSR's
>3FFF	16383	

BASIC MODE

ADDRESS HEX	ADDRESS DECIMAL	
>ØØØØ	Ø	SCREEN IMAGE TABLE
>Ø2FF	767	
>Ø3ØØ	768	SPRITE AND COLOR TABLE
>Ø31F	799	
>32Ø	8ØØ	BASIC BUFFER
>3BD	957	
>3BE	958	BASIC TEMPORARY AND INTERPRETER ROLLOUT
>Ø3FF	1Ø23	
>Ø4ØØ	1Ø24	PATTERN DESCRIPTOR TABLE
>Ø5FF	1535	
>Ø6ØØ	1536	VALUE STACK STRING SPACE DYNAMIC SYMBOL TABLE AND PAB's STATIC SYMBOL TABLE LINE NUMBER TABLE BASIC PROGRAM
>3FFF	16383	

Here is the sample program from Chapter Eight. This version incorporates the changes necessary for this assembly language program to be called by a BASIC program:

```
       DEF  GO
       REF  VWTR,VSBW,VMBW,KSCAN
WR     BSS  >2Ø
STATUS EQU  >837C
KEYVAL EQU  >8375
DTEN   DATA >A

*****************************************************************

* ALL SPACE CHARACTER CODES (>2Ø) HAVE BEEN INCREMENTED
* BY >6Ø

BORDER DATA >FFFF,>8Ø8Ø,>8Ø8Ø,>8Ø8Ø
       DATA >8Ø8Ø,>8Ø8Ø,>8Ø8Ø,>8Ø8Ø
       DATA >8Ø8Ø,>8Ø8Ø,>8Ø8Ø,>8Ø8Ø
       DATA >8Ø8Ø,>8Ø8Ø,>8Ø8Ø,>FFFF

*****************************************************************


MSG1   TEXT '** PRESS ANY KEY    **'
 SG2   TEXT '* KEYSTROKE VALUE IS *'
MSG3   TEXT '* PRESS REDO/ESCAPE  *'


*****************************************************************

* SCREEN BIAS OR OFFSET CONSTANT FOR BASIC


OFFST  BYTE >6Ø

*****************************************************************

REDOV  BYTE >Ø6
ESCV   BYTE >ØF
       EVEN       FORCE AN EVEN WORD BOUNDARY IN THE
                            LOCATION COUNTER
SAV11  BSS  2
GO     MOV  R11,@SAV11
       LWPI WR
       LI   RØ,>Ø755
       BLWP @VWTR
```

```
*****************************************************************

* THE ADDRESSES USED BY THESE INSTRUCTIONS TO ACCESS THE COLOR
* TABLE IN VDP RAM HAVE BEEN ADJUSTED FOR BASIC


       LI   RØ,799
       LI   R1,>55ØØ
       BLWP @VSBW
       LI   RØ,78Ø
       LI   R1,>1FØØ
CLOOP  BLWP @VSBW
       CI   RØ,798
       JEQ  BPUT
       INC  RØ
       JMP  CLOOP

*****************************************************************

BPUT   LI   RØ,Ø
       LI   R1,BORDER
       LI   R2,32
BLOOP  BLWP @VMBW
       CI   RØ,736
       JEQ  EXIT
       AI   RØ,32
       JMP  BLOOP
EXIT   LI   RØ,261      VDP RAM ADDRESS FOR FIRST MESSAGE
       LI   R2,MSG1     CPU RAM ADDRESS OF THE MESSAGE
       LI   R3,22       MESSAGE LENGTH
       BL   @PBASIC     PRINT MESSAGE
SCAN1  CLR  @STATUS
       BLWP @KSCAN
       MOVB @STATUS,@STATUS
       JEQ  SCAN1
       LI   RØ,325      VDP RAM ADDRESS FOR SECOND MESSAGE
       LI   R2,MSG2     CPU RAM ADDRESS OF THE MESSAGE
       LI   R3,22       MESSAGE LENGTH
       BL   @PBASIC     PRINT MESSAGE
       LI   RØ,395
       MOVB @KEYVAL,R1

*****************************************************************

* THE KEYSTROKE VALUE MUST BE ADJUSTED BY >6Ø BEFORE IT IS
* DISPLAYED

       AB   @OFFST,R1

*****************************************************************
```

```
       BLWP @VSBW
       CLR  R4
       MOVB @KEYVAL,R4
       SLR  R4,8
       LI   R3,404
       LI   RØ,406
       BL   @FIGUR
       LI   RØ,485        VDP RAM ADDRESS FOR THIRD MESSAGE
       LI   R2,MSG3       CPU RAM ADDRESS OF THE MESSAGE
       LI   R3,22         MESSAGE LENGTH
       BL   @PBASIC       PRINT MESSAGE
SCAN2  CLR  @STATUS
       BLWP @KSCAN
       MOVB @STATUS,@STATUS
       JEQ  SCAN2
       CB   @KEYVAL,@ESCV
       JEQ  ESCAP
       CB   @KEYVAL,@REDOV
       JNE  SCAN2
       B    @BPUT
FIGUR  MOV  R4,R5
       CLR  R4
       DIV  @DTEN,R4
       AI   R5,>ØØ3Ø
       SLA  R5,8
       MOV  R5,R1

*****************************************************************

* THE ASCII NUMERIC MUST BE ADJUSTED BY >6Ø BEFORE IT IS
* DISPLAYED

       AB   @OFFST,R1

*****************************************************************


       BLWP @VSBW
       DEC  RØ
       C    RØ,R3
       JHE  FIGUR
       RT
ESCAP  CLR  @STATUS
       MOV  @SAV11,R11
       RT
```

```
*****************************************************************

* "PBASIC" ROUTINE ADDS THE BASIC OFFSET TO EACH BYTE *
*         OF THE DISPLAY DATA AND WRITES ONE BYTE AT *
*         A TIME TO THE SCREEN                       *

*****************************************************************

PBASIC MOVB *R2+,R1       MOVE ONE BYTE OF MESSAGE TO R1
       AB   @OFFST,R1     ADJUST FOR BASIC
       BLWP @VSBW         WRITE ONE BYTE
       DEC  R3            DECREMENT CHARACTER COUNT
       JNE  PBASIC        IF NOT ZERO, DO IT AGAIN
       RT                 ELSE RETURN

*****************************************************************


       END
```

Now, to call this routine from a TI BASIC program, you could write a program in this format:

```
10 REM ***************************
20 REM * BASIC PROGRAM TO CALL   *
30 REM * THE CHAPTER EIGHT       *
40 REM * ASSEMBLY PROGRAM        *
50 REM ***************************
60 CALL INIT
70 CALL LOAD("DSK1.OBJECT")
80 CALL LINK("GO")
90 CALL SCREEN(4)
100 FOR LOOP=780 TO 799
110 CALL POKEV(LOOP,19)
120 NEXT LOOP
130 GOTO 80
```

For Mini-Memory users who have already assembled the program, stored it in the module and added the program's name and entry point to the REF/DEF table, the LOAD instruction at line 70 is not necessary. CALL INIT initializes memory, clears previously loaded programs and data and checks to see if memory expansion is attached and, if so, initializes the values needed to link memory expansion to console memory for assembly language use. CALL INIT must appear in your program before the first CALL LOAD.

CALL LOAD loads the object code from the device/file name specified. It can also be used to place (or "poke") values into CPU RAM with the format CALL LOAD(ADDRESS,VALUE1, VALUE2,VALUE3,...). CALL LINK branches to the entry point address in the assembly language program. Control is passed at this point to the subprogram. Execution of the remaining BASIC statements does not continue until the assembly program completes its task and returns.

The assembly program in this example resets the screen and character colors to white on blue. Upon returning from the subprogram, the BASIC program resets the screen and character colors back to black on green.

In addition to using CALL LOAD to place values into CPU RAM, other handy instructions are available in TI BASIC with Editor/Assembler and Mini-Memory. CALL PEEK(ADDRESS, VARIABLE1,VARIABLE2,...) allows you to retrieve one or more values from an address in CPU RAM. Similarly, CALL PEEKV(ADDRESS,VARIABLE1, VARIABLE2,...) retrieves a value from VDP RAM. CALL POKEV(ADDRESS,VALUE1, VALUE2,...) places one or more values into an address in VDP RAM.

The format for each of these statements is the same. Values are stated in decimal format. Addresses larger than 32767 are expressed in negative two's complement notation. Each variable or value denotes one byte of data. When more than one value or variable is used, the first is assigned to the byte at the address specified and each successive value or variable is assigned to the next byte at the next address.

The POKEV statement resets the foreground and background color values in the color table to black on green in the BASIC program example above. Each byte in the color table is set to >13 or decimal 19. With your knowledge of tables and addresses, using POKEs and PEEKs in your BASIC program can perform certain tasks faster than normal BASIC statements and also do things that are not normally possible with BASIC. Remember the screen offset or bias of 96 affects these statements as well. For example, CALL POKEV(2,161) displays an "A" at row one column three. Although the normal ASCII code of "A" is 65, adding the offset (65 + 96) makes this value 161.

CPU RAM bytes >837D, >837E and >837F are part of the area known as the CPU RAM PAD. These addresses make up the VDP character buffer. By placing a character code value at >837D, the character will be displayed on the screen at the row and column specified at >837E and >837F respectively. Since address >837D (decimal 33661) is greater than 32767, the address must be given in two's complement notation. Also, these addresses are in CPU RAM and can be accessed with CALL LOAD. In this way, some VDP access can be obtained without using POKEV, which is not available in Extended BASIC. CALL LOAD(-31875,161,1,3) displays "A" at row 1 column 3, just as the POKEV example did. The first value (161) is placed at address >837D (-31875), the second value (1) is placed at address >837E (-31874), and the third value (3) is placed at address >837F (-31873).

THE EXTENDED BASIC ENVIRONMENT

There are differences in how assembly subprograms may be used with TI Extended BASIC compared to TI BASIC. Particularly, the usage of VDP RAM varies from both the BASIC and Assembler mode. Also, CPU RAM, the area occupied by the memory expansion unit, can be used by Extended BASIC. Addresses >2000 to >3FFF and >A000 to >FFE0 are used by the Extended BASIC loader. Assembly language programs may only be up to 8K bytes in length with Extended BASIC.

VDP RAM UTILIZATION
EXTENDED BASIC MODE

ADDRESS		
HEX	DECIMAL	
>0000	0	SCREEN
		IMAGE
>02FF	767	TABLE
>0300	768	SPRITE
		ATTRIBUTE
>03FF	1023	LIST
>0400	1024	PATTERN
		DESCRIPTOR
		AND SPRITE
		DESCRIPTOR
>077F	1919	TABLE
>0780	1920	SPRITE
		MOTION
>07FF	2047	TABLE
>0800	2048	COLOR
>081F	2079	TABLE
>0820	2080	EXTENDED
		BASIC
		PROGRAM
		INTERPRETER,
		WORK AREAS,
		DATA TABLES,
>3FFF	16383	ETC.

The Extended BASIC loader does not recognize external references (REFs). For access to utilities such as VSBW, the addresses of the utilities must be EQUated to their names. These utilities are located at different addresses than they are in BASIC or assembler modes. Not all utilities are supported. For example, DSRLNK is not supported under Extended BASIC. When your assembly language program is run from Extended BASIC, or when it is run automatically by including the entry point label with the END directive, it starts out in the GPL workspace. The GPL workspace begins at >83E0. You must not use this area as your own workspace. Rather, you should establish your own workspace by defining a workspace and using the LWPI instruction at the start of your program. This has been done in every program example so far and generally makes for good practice. In order for your program to properly return to the program which called it, you should set the workspace pointer register so that it points to the GPL workspace before returning. The following instructions illustrate how this is done.

```
GPLWS  EQU  >83E0         BEGINNING ADDRESS OF THE GPL WORKSPACE
SAV11  BSS  2             SAVE AREA FOR RETURN ADDRESS
MYWS   BSS  >20           SET ASIDE 32 BYTES FOR MY WORKSPACE
•
•
START  MOV  R11,@SAV11    SAVE RETURN ADDRESS
       LWPI MYWS          ESTABLISH MY OWN WORKSPACE
•
•
END    LWPI GPLWS         RE-ESTABLISH THE GPL WORKSPACE
       MOV  @SAV11,R11    RESTORE RETURN ADDRESS
       CLR  STATUS        CLEAR THE GPL STATUS BYTE
       RT                 RETURN
```

Another way of ending the program works in any environment, no matter if your assembly program is called from BASIC or Extended BASIC. For this method, the contents of R11 do not need to be saved. The following instructions illustrate this technique.

```
GPLWS  EQU  >83E0
•
•
END    LWPI GPLWS
       CLR  @STATUS
       B    @>0070
```

Once again, here is the sample program from Chapter Eight. This time, it has been written with the changes necessary to call it from Extended BASIC.

```
       DEF  GO

*********************************************************
* THE EXTENDED BASIC LOADER DOES NOT RECOGNIZE EXTERNAL *
* REFERENCES. THE UTILITIES VWTR, VSBW, VMBW, AND KSCAN *
* MUST BE ACCESSED BY INCLUDING EQUATES FOR THEM.       *
* THE ADDRESSES OF THE UTILITIES ARE DIFFERENT UNDER    *
* EXTENDED BASIC                                        *
*********************************************************

VWTR   EQU  >2030
VSBW   EQU  >2020
VMBW   EQU  >2024
KSCAN  EQU  >201C

*********************************************************
* "GPLWS" address of the GPL workspace                  *
*********************************************************

GPLWS  EQU  >88E0
WR     BSS  >20
STATUS EQU  >837C
KEYVAL EQU  >8375
DTEN   DATA >A
BORDER DATA >FFFF,>8080,>8080,>8080
       DATA >8080,>8080,>8080,>8080
       DATA >8080,>8080,>8080,>8080
       DATA >8080,>8080,>8080,>FFFF
MSG1   TEXT '** PRESS ANY KEY     **'
MSG2   TEXT '* KEYSTROKE VALUE IS  *'
MSG3   TEXT '* PRESS REDO/ESCAPE   *'
OFFST  BYTE >60
REDOV  BYTE >06
ESCV   BYTE >OF
       EVEN
GO     LWPI WR
       LI   R0,0755
       BLWP @VWTR
```

```
*********************************************************
* THE ADDRESSES USED BY THESE INSTRUCTIONS TO ACCESS    *
* THE COLOR TABLE HAVE BEEN CHANGED                     *
*********************************************************


       LI   RØ,>Ø81F
       LI   R1,>55ØØ
       BLWP @VSBW
       LI   RØ,>Ø8ØØ
       LI   R1,>1FØØ
CLOOP  BLWP @VSBW
       CI   RØ,>Ø81E
       JEQ  BPUT
       INC  RØ
       JMP  CLOOP
BPUT   LI   RØ,Ø
       LI   R1,BORDER
       LI   R2,32
BLOOP  BLWP @VMBW
       CI   RØ,736
       JEQ  EXIT
       AI   RØ,32
       JMP  BLOOP
EXIT   LI   RØ,261        VDP RAM ADDRESS FOR FIRST MESSAGE
       LI   R2,MSG1       CPU RAM ADDRESS OF THE MESSAGE
       LI   R3,22         MESSAGE LENGTH
       BL   @PBASIC       PRINT MESSAGE
SCAN1  CLR  @STATUS
       BLWP @KSCAN
       MOVB @STATUS,@STATUS
       JEQ  SCAN1
       LI   RO,325        VDP RAM ADDRESS FOR SECOND MESSAGE
       LI   R2,MSG2       CPU RAM ADDRESS OF THE MESSAGE
       LI   R3,22         MESSAGE LENGTH
       BL   @PBASIC       PRINT MESSAGE
       LI   RØ,395
       MOVB @KEYVAL,R1
       AB   @OFFST,R1
       BLWP @VSBW
       CLR  R4
       MOVB @KEYVAL,R4
       SRL  R4,8
       LI   R3,4Ø4
       LI   RØ,4Ø6
       BL   @FIGUR
       LI   RØ,485        VDP RAM ADDRESS FOR THIRD MESSAGE
       LI   R2,MSG3       CPU RAM ADDRESS OF THE MESSAGE
       LI   R3,22         MESSAGE LENGTH
       BL   @PBASIC       PRINT MESSAGE
```

```
SCAN2  CLR  @STATUS
       BLWP @KSCAN
       MOVB @STATUS,@STATUS
       JEQ  SCAN2
       CB   @KEYVAL,@REDOV
       JNE  SCAN2
       B    @BPUT
FIGUR  MOV  R4,R5
       CLR  R4
       DIV  @DTEN,R4
       AI   R5,>ØØ3Ø
       SLA  R5,8
       MOV  R5,R1
       AB   @OFFST,R1
       BLWP @VSBW
       DEC  RØ
       C    RØ,R3
       JHE  FIGUR
       RT




*******************************************************
*                                                     *
*  "ESCAP" RESETS THE WORKSPACE POINTER REGISTER TO   *
*          THE GPL WORKSPACE, CLEARS THE STATUS BYTE, *
*          AND BRANCHES TO ADDRESS >ØØ7Ø              *
*                                                     *
*******************************************************

ESCAP  LWPI GPLWS
       CLR  @STATUS
       B    @>ØØ7Ø
PBASIC MOVB *R2+,R1          MOVE ONE BYTE OF MESSAGE TO R1
       AB   @OFFST,R1        ADJUST FOR BASIC
       BLWP @VSBW            WRITE ONE BYTE
       DEC  R8               DECREMENT CHARACTER COUNT
       JNE  PBASIC           IF NOT ZERO DO IT AGAIN
       RT                    ELSE RETURN
       END
```

```
10 REM ****************************
20 REM * EXT BASIC PROGRAM TO CALL *
30 REM * THE CHAPTER EIGHT         *
40 REM * ASSEMBLY PROGRAM          *
50 REM ****************************
60 CALL INIT
70 CALL LOAD("DSK1.OBJECT")
80 CALL LINK("GO")
90 CALL SCREEN(4)
100 FOR LOOP=0 TO 16
110 CALL COLOR(LOOP,2,4)
120 NEXT LOOP
130 GOTO 80
```

In addition to having BASIC and Extended BASIC programs call assembly language programs, you may pass numeric and string data from one to the other. This may be done with PEEKs and POKEs, or the CALL LINK statement can be used. The entry point address in CALL LINK can be followed by up to 16 variables which are available to the assembly language for it to act upon.

The following subprogram example emulates the DISPLAY AT statement in Extended BASIC. It is written to be run from TI BASIC with either the Mini-Memory or Editor/Assembler module installed. The program was written with Editor/Assembler, so if you will be using the Line-by-Line assembler with the Mini-Memory, you will need to make the changes mentioned In Chapter Five. A similar program for the Line-by-Line assembler is listed in the Mini-Memory manual.

The format of a TI BASIC program that will use this subprogram will vary depending on which module is used and which hardware configuration you use with your computer. For Mini-Memory users who have already assembled the program, stored it in the module and added the program name and entry point to the REF/DEF table, this format would be used:

CALL LINK('DEF LABEL",ROW,COL,STRING)

For example:

110 CALL LINK('GO",12,6,S$)

Editor/Assembler users would use a format similar to this TI BASIC program:

```
10 REM *****************************
20 REM * BASIC PROGRAM TO CALL     *
30 REM * THE "DISPLAY AT" ASSEMBLY *
40 REM * LANGUAGE SUBPROGRAM       *
50 REM *****************************
60 CALL INIT
70 CALL LOAD("DSK1.BSCSUP")
80 CALL LOAD("DSK1.OBJECT")
90 INPUT "STRING?":S$
100 INPUT "ROW?":R
110 INPUT "COLUMN?":C
120 CALL LINK("GO",R,C,S$)
130 FOR DELAY=1 TO 500
140 NEXT DELAY
150 GOTO 90
```

The special routines used in this assembly program known as "Basic Support" (NUMREF,STRREF,ERR) are included on diskette "A" of the Editor/Assembler package. These must be loaded into the computer with a BASIC statement, such as the CALL LOAD("DSK1.BSCSUP") statement used here, in order for the assembly program to resolve the symbolic names of these routines. These routines already exist in the Mini-Memory module, so no such statement is required. Next, the object file which was created by assembling the program is loaded. You should specify the device name that fits your situation.

```
** "DISPLAY AT" SUBPROGRAM **

           DEF  GO
           REF  VSBW,VMBW,VMBR,NUMREF,XMLLNK,STRREF,ERR
   FPAC    EQU  >834A
   SBUFF   BSS  256            DEFINE STRING BUFFER
   LBUF    BSS  32             DEFINE LINE BUFFER
   LIM     DATA >0001,>0018,>001C   DEFINE ROW & COLUMN RANGES
   GO      MOV  R11,R10        SAVE RETURN ADDRESS
           CLR  RO             CLEAR RO - NUMREF PARAM #1
           LI   R1,1           INIT R1 TO 1 - NUMREF PARAM #2
           BL   @GETNUM        GET 1ST VARIABLE (ROW NUMBER)
           BL   @CHKLMR        CHECK ROW LIMITS
           MOV  @FPAC,R4       MOVE ROW VALUE TO R4
           DEC  R4             ADJUST FOR ASSEMBLY LANGUAGE
           SLA  R4,5           MULTIPLY ROW BY 32
           MOV  R4,R7          MOVE ROW ADDR TO R7
           INC  R1             INCREMENT R1 FOR NEXT PARAMETER
           BL   @GETNUM        GET 2ND VARIABLE (COLUMN NUMBER)
           BL   @CHKLMC        CHECK COLUMN LIMITS
           A    @FPAC,R4       ADD IN ROW VALUE
```

```
        INC  R4           ADJUST FOR BASIC
        INC  R1           INCREMENT R1 FOR NEXT PARAMETER
        LI   R2,SBUF      LOAD R2 W/STRING BUFFER ADDRESS
        SETO @SBUF        INITIALIZE FIRST WORD TO >FFFF
        BLWP @STRREF      GET 3RD VARIABLE (THE STRING)
        CLR  R5           CLEAR R5 (BYTE COUNTER)
        MOV  R2,R3        MOVE SBUF ADDRESS TO R3
        MOVB *R3+,R5      GET 1ST BYTE (STRING LENGTH)
        JEQ  XT           IF STRING LENGTH EQUALS ZERO EXIT
        SWPB R5           SWAP THE BITS OF R5 (LEFT ADJ)
        BL   @PRINT       DISPLAY THE STRING
XT      B    *RIO         RETURN TO CALLING PROGRAM
GETNUM  BLWP @NUMREF      GET BASIC NUMBER (FLOATING POINT)
        BLWP @XMLLNK      PERFORM FLTP TO INT CONVERSION
        DATA >12ØØ        ADDRESS OF XML ROUTINE
        B    *R11         RETURN
CHKLMC  C    @FPAC,@LIM+4 COMPARE THE INTEGER TO 28
        JGT  ERROR        IF GREATER THAN, JUMP TO "ERROR"
        JMP  CHK          JUMP TO "CHK"
CHKLMR  C    @FPAC,@LIM+2 COMPARE THE INTEGER TO 24
        JGT  ERROR        IF GREATER THAN, JUMP TO "ERROR"
CHK     C    @FPAC,@LIM   COMPARE THE INTEGER TO 1
        JLT  ERROR        IF LESS THAN, JUMP TO "ERROR"
        B    *R11         RETURN
ERROR   LI   Ø,>13ØØ      LOAD RO WITH ERROR MESSAGE VALUE
        BLWP @ER R        BRANCH TO ERROR MESSAGE ROUTINE
PRINT   MOV  R11,9        SAVE LINKAGE ADDRRESS
        LI   R6,>6ØØØ     LOAD R6 WITH SCREEN OFFSET VALUE
        AI   R7,3Ø        CALC NEXT ROW ADDRESS
PLOOP   MOV  R4,RØ        MOVE VDP ADDRESS TO RO
        MOVB *R3+,R1      GET A BYTE FROM STRING BUFFER
        AB   R6,R1        ADD SCREEN OFFSET TO R1
        BLWP @VSBW        WRITE ONE BYTE
        INC  R4           POINT TO NEXT SCREEN ADDRESS
        DEC  R5           DECREMENT THE CHARACTER COUNT
        JNE  L1           IF R5 IS NOT = Ø, JUMP TO L1
        B    *R9          RETURN
L1      C    R4,R7        COMPARE NEW ADDRESS TO LIMIT
        JL   PLOOP        IF LESS THAN, JUMP TO "PLOOP"
        AI   R7,32        ELSE INC LIMIT BY 1 ROW (32)
        Al   R4,4         INCREMENT SCREEN ADDRESS BY 4
        CI   R7,766       IS NEW ROW OFF THE SCREEN
        JLE  PLOOP        JUMP TO "PLOOP"
        BL   @SCROLL      BRANCH TO SCROLL
        AI   R7,-32       ADJUST LIMIT AFTER SCROLL
        AI   R4,-32       ADJUST SCREEN ADDRESS AFTER SCROLL
        JMP  PLOOP        JUMP TO "PLOOP"
SCROLL  LI   RO,-32       INITIALIZE SCREEN ADDRESS
        LI   R1,LBUF      LOAD R1 W/LINE BUFFER ADDRESS
        LI   R2,32        LOAD R2 WITH LENGTH OF LINE
```

```
7Ø L4      AI    RO,64        MOVE DOWN ONE LINE
71         BLWP  @VMBR        READ A LINE INTO LINE BUFFER
72         AI    RO,-32       MOVE UP ONE LINE
73         CI    RO,>2EØ      IS THIS THE LAST LINE?
74         JLT   NP           IF NOT JUMP TO "NP"
75         JEQ   S1           IF IT IS, JUMP TO "S1"
76         B     *R11         SCROLL IS DONE, RETURN
77 S1      MOV   R1,R13       COPY BUFFER POINTER
78         MOV   R2,R14       COPY BUFFER LEN
79         LI    R15,>2Ø2Ø    LOAD 2 SPACES INTO R15
8Ø L3      MOV   R15,*R13+    MOVE SPACES TO BUFFER
81         DECT  R14          DECREMENT BYTE COUNT
82         JNE   L3           PAD NEXT WORD
83 NP      BLWP  @VMBW        WRITE MULTIPLE BYTES
84         JMP   L4           JUMP TO "L4"
85         END
```

Lines 8 and 9 clear R0 and load R1 with the value of 1. These registers are used by the NUMREF and STRREF routines as indicators. The value of R0 tells the routine what type of value it will retrieve. The value of R1 tells the routines which variable is to be retrieved. A "1" indicates the first value passed from the BASIC program by the CALL LINK statement. The first value in the example is the row number. Line 10 performs a Branch and Link to the routine "GETNUM" at line 32. NUMREF gets the value (which is in floating point format) from the TI BASIC value stack and places it at address >834A. This address is the Floating Point Accumulator. Next, the XMLLNK routine takes the floating point number and converts it to an integer (actually a binary expression which the assembly program can act upon). The result of the conversion is still at the address of the Floating Point Accumulator (>834A).

It must be verified that this number is a valid row number from 1 to 24. Line 11 performs a Branch and Link to the routine "CHKLMR" at line 39. First the value is compared to the value at LIM + 2, which is 24. Then the value is compared to the value at LIM which is 1. If the row value is greater than 24 or less than 1, then a jump is made to the label ERROR at line 44. The error routine causes the message "BAD VALUE" to be displayed and returns to the BASIC program.

Line 12 in the assembly program is only reached when the row value has been verified as valid. Line 12 moves the row value into R4. The value needed to calculate the screen address in assembly language terms for this row is one less than the actual value retrieved. Line 13 adjusts for this by DECrememting R4. The screen address of this row is equal to the value now in R4 multiplied by 32. Line 14 accomplishes this multiplication by shifting the bits of R4 to the left 5 positions. Any time the bits of a register are shifted left, the effect on the value of the register is equal to VALUE * 2 ∧ N, where N is the number of positions shifted to the left. Since R4 is shifted left 5 positions and 2 ∧ 5 = 32, the effect is the same as multiplying by 32. Using the shift left instruction this way is a handy way to perform multiplication if the value you wish to multiply by is a power of 2. The value now in R4 is the screen address of the selected row. Line 15 saves this value in R7.

Line 16 adds 1 to the value in R1 before lines 17 and 18 retrieve and verify the column number. Adding one to R1 tells the routines NUMREF and STRREF to operate on the next value which was passed by CALL LINK. Line 19 is only reached once the column number has been retrieved and verified. It adds the column value to the row address in R4. Line 20 adds one to this value to make it correspond to the way in which DISPLAY AT handles row and column addresses. Ordinarily, when the user selects row one, column one, this translates to screen image table address 0, the first position on the screen. However, DISPLAY AT only recognizes columns 3 through 30 in each row for a length of 28 bytes per row. Thus, if row one, column one is selected, the actual display will begin at row one, column three. Columns 1, 2, 31 and 32 of each row contain filler characters and are not used for displays. Register four now contains the VDP RAM screen image table address which corresponds to the row and column numbers selected.

At line 21, R1 is again incremented to indicate to STRREF that it is to operate on the next value that was passed by CALL LINK. The STRREF utility retrieves the BASIC string and converts it to an assembly language string. Line 22 loads R2 with the address of where the string is to end up (SBUF). The first byte of the string buffer must be a value which sets the maximum number of bytes to be accepted. When the string is returned by STRREF it will start at byte SBUF+1, and the first byte (SBUF+0) will be changed to reflect the actual length of the string. The maximum number of bytes to be accepted is 255, or >FF. Line 23 uses the SETO instruction, which sets all the bits in the named address to ones. This is the opposite effect as using CLR, which sets all bits to zero. If all the bits of a word in memory are set to ones, the hex value of that word is >FFFF. Line 24 performs the BLWP to STRREF, which actually gets the string. Line 26 moves the address of the string buffer to R3 since R2 will be used later. Line 27 moves the string length found at the first byte of the string buffer to R5, which was cleared at line 25. Line 27 uses the auto-increment addressing feature, leaving the address in R3 pointing to SBUF+1 after the move. If the string length is zero when moved into R5, the equal bit will be turned on in the status register. This condition is checked at line 28, which directs the logic to return to the calling program, since a length of zero indicates that no string has been passed. Line 29 uses the Swap Bytes (SWPB) instruction to change the value in R5 from the left byte to the right byte. If the value moved into R5 at line 27 was >FF, then R5 would contain >FF00. After the SWPB instruction, R5 would contain >00FF, or decimal 255. Line 30 performs a Branch and Link to the routine "PRINT" which will display the string at the desired screen location.

Line 47 loads R6 with the Screen offset value which must be added to any ASCII character code before it can be properly displayed by an assembly language program which has been called from BASIC. Line 48 adds 30 to the value in R7. The value in R7 was the screen address of the row selected. R7 now contains the screen address value of the 31st column of the row selected. If the new screen address in R4 has reached this value, then it is necessary to move to the next row before continuing with the display of the string.

Line 49 moves the current screen address in R4 to R0. Line 50 moves a byte from the string buffer to R1 and auto-increments the address in R3 by one. Line 51 adds the screen offset value (>60) to R1 before line 52 writes it to the screen with the VDP Single Byte Write (VSBW) routine. Lines 53 and 54 increment the screen address and decrement the character count after each write. If the value in R5 reaches zero, the equal bit is turned on in the status register. Line 55 checks for this condition and directs the logic of the routine to label "L1" as long as R5 is not zero. If R5 is equal to zero, then all the characters of the string have been displayed, and Line 56 returns to the program.

Line 57 compares the new screen address to be used with the value in R7. If R4 is greater than or equal to R7, the next byte to be displayed should go on the next row starting at column three. If R4 is less than R7, line 58 directs the logic to the label "PLOOP" to write another byte. Lines 59 and 60 adjust the screen address and row limit values for the next row, column three. Line 61 compares the row limit value to 766. The maximum allowable row that could have been selected is 24. The address of the first column of row 24 is equal to 736. Adding the row limit value to this gives the maximum value for R7 of 766. If R7 is greater than or equal to 766, the current screen display must be scrolled up before the rest of the string can be displayed. As long as this maximum value has not been reached, line 62 will jump to the label "PLOOP" to write another byte. Line 63 performs a Branch and Link to SCROLL when the maximum value in R7 has been reached.

The effect of scrolling the screen display is to move all of the values in the screen table up one row and fill the bottom row with spaces. Anything displayed on the first row will be lost as it scrolls off the screen. To aid in this, a temporary storage buffer (LBUF) was set aside at line 5 to hold one row of data (32 bytes). Lines 67, 68 and 69 load the initial values needed for the scroll operation. The VDP RAM address value in R0 is initially set -32 because line 70 will add 64 to it each time through the loop named L4. The first time through this loop, R0 will become equal to 32, the address of row two. R1 must contain the address of the line buffer and R2 the length, 32 bytes.

Line 71 reads one row of characters from the screen into the line buffer (LBUF). Line 72 adjusts the VDP RAM address in R0 by 32 and line 73 checks the new value to see if SCROLL is done. If the value in R0 is equal to >2E0, the logic jumps to label S1. Otherwise, logic continues at NP at line 83. Line 83 writes the contents of the line buffer to the new screen address and line 84 jumps to the label L4 to complete the loop. This loop continues until the compare at line 73 is true at which time logic jumps to the label S1. Lines 77 through 82 copy the line buffer address and length into registers R13 and R14 and proceeds to fill the line buffer with spaces (>20). Line 82 keeps returning to label L3 until all 32 bytes of the line buffer have been filled with spaces. The logic of the routine then falls to line 83 which writes the contents of the line buffer (all spaces) to the last row on the screen.

Logic transfers to label L4. The Compare at line 73 now has no effect on the logic of SCROLL, and line 76 returns to line 64 of PRINT. After scrolling, the limit and screen address values are adjusted by -32 at lines 64 and 65 before continuing to display the rest of the string.

Included here are two programs that demonstrate the use of assembly routines with TI BASIC or TI Extended BASIC.

ASSEMBLER CHARACTER DEFINITION

This program redefines the standard character set so that lower case letters appear as "true" lower case with descenders. This means the tails of the p,q,g,j and y can be printed on the screen. Also, some special foreign language characters have been defined. The Data statements in this routine use the same type of 16-character code used to define characters with the CALL CHAR statement in TI BASIC. Characters with ASCII codes 30 through 126 have definitions provided. Space for data to redefine characters up to ASCII 143 has been included so that you may use the routine to quickly define graphics characters at the beginning of a TI BASIC or TI Extended BASIC program.

Note that the program cannot be assembled if the source file contains both the Extended BASIC DEF/EQU and the Editor/Assembler DEF/REF. To use the program with both TI BASIC and Extended BASIC, save separate versions of the source code and assemble two object files on separate disks. The Extended BASIC version should be assembled using the "R" option of Editor/Assembler, while the BASIC version should be assembled with the "C" option. The object file name should be "DSK1.CHARDF".

To call the routine from your TI BASIC or Extended BASIC program, you need to use the following routine at the beginning of the program.

```
100 CALL INIT
110 CALL CLEAR
120 CALL LOAD("DSK1.CHARDF")
130 CALL LINK("CHARDF")
```

If you use Extended BASIC, you can save the above routine as the filename "DSK1.LOAD". When you select Extended BASIC, it will run automatically. Then, any Extended BASIC program you rbn will have the characters defined by the routine if you use a CALL LINK("CHARDF") statement. Once the routine is loaded into memory expansion, it needn't be reloaded unless memory expansion is turned off.

```
* ASSEMBLER CHARACTER DEFINITION *
*     by David Migicovsky         *
*     Copyright (C) 1983          *
*  by Steve Davis Publishing      *

* THE NEXT TWO LINES ARE FOR USE WITH
* TI EXTENDED BASIC VERSION ONLY

        DEF CHARDF,VMBW

 VMBW   EQU >2024
```

```
* THE NEXT TWO LINES ARE FOR USE WITH
* TI BASIC AND EDITOR/ASSEMBLER ONLY

       DEF CHARDF

       REF VMBW

*             PATTERN IDENTIFIERS      *CHAR ASCII

NEWDEF DATA >7C7C,>6C6C,>6C6C,>7C7C  *       3Ø
       DATA >ØØØØ,>ØØØØ,>ØØØØ,>ØØØØ  *       31
       DATA >ØØØØ,>ØØØØ,>ØØØØ,>ØØØØ  * " "  32
       DATA >1Ø1Ø,>1Ø1Ø,>1ØØØ,>1ØØØ  * "!"  33
       DATA >2828,>28ØØ,>ØØØØ,>ØØØØ  * """  34
       DATA >2828,>7C28,>7C28,>28ØØ  * "#"  35
       DATA >3854,>5Ø38,>1454,>38ØØ  * "$"  36
       DATA >6Ø64,>Ø81Ø,>2Ø4C,>ØCØØ  * "%"  37
       DATA >2Ø5Ø,>5Ø2Ø,>5448,>34ØØ  * "&"  38
       DATA >Ø8Ø8,>1ØØØ,>ØØØØ,>ØØØØ  * "'"  39
       DATA >Ø81Ø,>2Ø2Ø,>2Ø1Ø,>Ø8ØØ  * "("  4Ø
       DATA >2Ø1Ø,>Ø8Ø8,>Ø81Ø,>2ØØØ  * ")"  41
       DATA >ØØ28,>1Ø7C,>1Ø28,>ØØØØ  * "*"  42
       DATA >ØØ1Ø,>1Ø7C,>1Ø1Ø,>ØØØØ  * "+"  43
       DATA >ØØØØ,>ØØØØ,>ØØ3Ø,>1Ø2Ø  * ","  44
       DATA >ØØØØ,>ØØ7C,>ØØØØ,>ØØØØ  * "-"  45
       DATA >ØØØØ,>ØØØØ,>ØØ3Ø,>3ØØØ  * "."  46
       DATA >ØØØ4,>Ø81Ø,>2Ø4Ø,>ØØØØ  * "/"  47
       DATA >3844,>4444,>4444,>38ØØ  * "Ø"  48
       DATA >1Ø3Ø,>1Ø1Ø,>1Ø1Ø,>38ØØ  * "1"  49
       DATA >3844,>Ø4Ø8,>1Ø2Ø,>7CØØ  * "2"  5Ø
       DATA >3844,>Ø418,>Ø444,>38ØØ  * "3"  51
       DATA >Ø818,>2848,>7CØ8,>Ø8ØØ  * "4"  52
       DATA >7C4Ø,>78Ø4,>Ø444,>38ØØ  * "5"  53
       DATA >182Ø,>4Ø78,>4444,>38ØØ  * "6"  54
       DATA >7CØ4,>Ø81Ø,>2Ø2Ø,>2ØØØ  * "7"  55
       DATA >3844,>4438,>4444,>38ØØ  * "8"  56
       DATA >3844,>443C,>Ø4Ø8,>3ØØØ  * "9"  57
       DATA >ØØ3Ø,>3ØØØ,>3Ø3Ø,>ØØØØ  * ":"  58
       DATA >ØØØØ,>3Ø3Ø,>ØØ3Ø,>1Ø2Ø  * ";"  59
       DATA >Ø81Ø,>2Ø4Ø,>2Ø1Ø,>Ø8ØØ  * "<"  6Ø
       DATA >ØØØØ,>7CØØ,>7CØØ,>ØØØØ  * "="  61
       DATA >2Ø1Ø,>Ø8Ø4,>Ø81Ø,>2ØØØ  * ">"  62
       DATA >3844,>Ø4Ø8,>1ØØØ,>1ØØØ  * "?"  63
       DATA >3844,>5C54,>5C4Ø,>38ØØ  * "@"  64
       DATA >3844,>447C,>4444,>44ØØ  * "A"  65
       DATA >7824,>2438,>2424,>78ØØ  * "B"  66
       DATA >3844,>4Ø4Ø,>4Ø44,>38ØØ  * "C"  67
       DATA >7824,>2424,>2424,>78ØØ  * "D"  68
       DATA >7C4Ø,>4Ø78,>4Ø4Ø,>7CØØ  * "E"  69
       DATA >7C4Ø,>4Ø78,>4Ø4Ø,>4ØØØ  * "F"  7Ø
       DATA >3C40,>405C,>4444,>3800  * "G"  71
       DATA >4444,>447C,>4444,>4400  * "H"  72
       DATA >3810,>1010,>1010,>3800  * "I"  73
       DATA >0404,>0404,>0444,>3800  * "J"  74
       DATA >4448,>5060,>5048,>4400  * "K"  75
       DATA >4040,>4040,>4040,>7C00  * "L"  76
       DATA >446C,>5454,>4444,>4400  * "M"  77
       DATA >4464,>6454,>4C4C,>4400  * "N"  78
       DATA >7C44,>4444,>4444,>7C00  * "O"  79
       DATA >7844,>4478,>4040,>4000  * "P"  80
       DATA >3844,>4444,>5448,>3400  * "Q"  81
       DATA >7844,>4478,>5048,>4400  * "R"  82
       DATA >3844,>4038,>0444,>3800  * "S"  83
       DATA >7C10,>1010,>1010,>1000  * "T"  84
       DATA >4444,>4444,>4444,>3800  * "U"  85
       DATA >4444,>4428,>2810,>1000  * "V"  86
       DATA >4444,>4454,>5454,>2800  * "W"  87
       DATA >4444,>2810,>2844,>4400  * "X"  88
       DATA >4444,>2810,>1010,>1000  * "Y"  89
       DATA >7C04,>0810,>2040,>7C00  * "Z"  90
       DATA >0810,>3844,>7C40,>3800  * "["  91
       DATA >3030,>3FFF,>FE7C,>180C  * "\"  92
       DATA >2010,>3844,>7C40,>3800  * "]"  93
       DATA >3844,>4040,>4438,>1000  * "^"  94
       DATA >1028,>0038,>4848,>3400  * "_"  95
       DATA >0000,>3840,>4038,>1000  * "`"  96
       DATA >0000,>3848,>4848,>3400  * "a"  97
       DATA >6020,>3824,>2424,>7800  * "b"  98
       DATA >0000,>3844,>4044,>3800  * "c"  99
       DATA >0C08,>3848,>4848,>3C00  * "d"  100
       DATA >0000,>3844,>7C40,>3800  * "e"  101
       DATA >1824,>2070,>2020,>2000  * "f"  102
       DATA >0000,>3C44,>3C04,>0438  * "g"  103
       DATA >6020,>2834,>2424,>2400  * "h"  104
       DATA >1000,>7010,>1010,>7C00  * "i"  105
       DATA >0800,>1808,>0848,>4830  * "j"  106
       DATA >2020,>2428,>3028,>2400  * "k"  107
       DATA >3010,>1010,>1010,>7C00  * "l"  108
       DATA >0000,>A854,>5454,>5400  * "m"  109
       DATA >0000,>5824,>2424,>2400  * "n"  110
       DATA >0000,>3844,>4444,>3800  * "o"  111
       DATA >0000,>7824,>2438,>2020  * "p"  112
       DATA >0000,>3048,>3808,>080C  * "q"  113
       DATA >0000,>5824,>2020,>2000  * "r"  114
       DATA >0000,>3C40,>3804,>7800  * "s"  115
       DATA >2020,>7820,>2024,>1800  * "t" 116
       DATA >0000,>4848,>4848,>3400  * "u" 117
       DATA >0000,>4444,>2828,>1000  * "v" 118
       DATA >0000,>D454,>5454,>2800  * "w" 119
       DATA >0000,>4428,>1028,>4400  * "x" 120
       DATA >0000,>4444,>3C04,>0418  * "y" 121
       DATA >0000,>7C48,>1024,>7C00  * "z" 122
       DATA >1820,>2040,>2020,>1800  * "{" 123
       DATA >1010,>1000,>1010,>1000  * "|" 124
       DATA >3008,>0804,>0808,>3000  * "}" 125
       DATA >0000,>2054,>0800,>0000  * "~" 126
       DATA >0000,>0000,>0000,>0000  *     127
       DATA >0000,>0000,>0000,>0000  *     128
       DATA >0000,>0000,>0000,>0000  *     129
       DATA >0000,>0000,>0000,>0000  *     130
       DATA >0000,>0000,>0000,>0000  *     131
       DATA >0000,>0000,>0000,>0000  *     132
       DATA >0000,>0000,>0000,>0000  *     133
       DATA >0000,>0000,>0000,>0000  *     134
       DATA >0000,>0000,>0000,>0000  *     135
       DATA >0000,>0000,>0000,>0000  *     136
       DATA >0000,>0000,>0000,>0000  *     137
       DATA >0000,>0000,>0000,>0000  *     138
       DATA >0000,>0000,>0000,>0000  *     139
       DATA >0000,>0000,>0000,>0000  *     140
       DATA >0000,>0000,>0000,>0000  *     141
       DATA >0000,>0000,>0000,>0000  *     142
       DATA >0000,>0000,>0000,>0000  *     143

CHARDF LI R0,1008 *THIS IS A DECIMAL NUMBER
       LI R1,NEWDEF
       LI R2,904  *THIS IS A DECIMAL NUMBER
       BLWP @VMBW
       RT
       END
```

MINI-MEMORY CHARACTER DEFINITION

To use with the Line-by-Line Assembler and the Mini-Memory, delete all lines except the DATA statements. Change the label on the first line of DATA from "NEWDEF" to "ND". Insert two new lines at the beginning of the program, making the first DATA statement line number three:

```
Line 1              AORG   >7DØØ
Line 2     VM       EQU    >6Ø28
Line 3     ND       DATA   >7C7C .....
```

Immediately following the last line of DATA statements, add the following lines:

```
           CD       LI   RØ,1ØØ8
                    LI   R1,ND
                    LI   R2,9Ø4
                    BLWP @VM
                    CLR  @>837C
                    B    *11
                    END
```

To access the routine from a TI BASIC program, you would use the statement CALL LINK("CD").

BAR GRAPH PROGRAM

This program example was created by Phil West and Bernle Elsner, two 99/4 enthuslasts from "down under" in Australla. It Is a routine that is deslgned to be called from any TI BASIC program with the Mini-Memory or Editor/Assembler modules Installed.
The routine allows you to plot high-resolution bar graphs in various colors. To access the routine, use the TI BASIC statement CALL LINK("BGRAPH",COLUMN,COLOR,HEIGHT). COLUMN should be a number between 1 and 28, the screen column in which the bar is to appear. COLOR is the color of the bar, a number from 1 to 7, derived as follows:

Color Number	Color	Character Set Used
1	Black	10
2	Dark Blue	11
3	Dark Red	12
4	Dark Yellow	13
5	Dark Green	14
6	Magenta	15
7	White	16

HEIGHT should be a number between 1 and 160, Indicating the number of pixel rows tall that the bar wlll be. All bars are drawn beglnnlng at character row 20 and may extend upward as high as the first character row. Thls leaves character rows 21 through 24 open for text display. Following the assembly listing is a demonstration program in TI BASIC that allows you to see the results. If you will be using the Editor/Assembler module, you will need to load the BASIC Support file from the Editor/Assembler disk A. The demonstration program prompts you to put the disk in drive one. The object code of the Bar Graph assembly program should be saved with the filename of "BGRAPH/O" (for Bar Graph Object) and should be in disk drive one when running the demo.

```
*  BAR-GRAPH ROUTINE
*  FOR USE WITH MINI-MEMORY FROM TI BASIC
*  BY PHIL WEST AND BERNIE ELSNER
*
       DEF  BGRAPH
       REF  VMBW,VSBW,NUMREF,XMLLNK,ERR
*
D1     DATA >0000         *
       DATA >0000         *
       DATA >0000         *
       DATA >003C         *   CHARACTER DEFINITIONS
       DATA >3C3C         *
       DATA >3C3C         *
       DATA >3C3C         *
       DATA >3C3C         *
```

```
D2      DATA >1040          *
        DATA >60A0          *   COLOUR BYTES
        DATA >C0D0          *
        DATA >F000          *
BGRAPH CLR  R0
*
*  GET THREE PARAMETERS FROM LINK LIST
*
        LI   R1,>0001       * GET FIRST PARAMETER
        BLWP @NUMREF
        BLWP @XMLLNK
        DATA >1200
        MOV  @>834A,R3
        CI   R3,>0000       * CHECK IF VALID VALUE
        JGT  C
        B    @E
C       CI   R3,>001D
        JLT  F
        B    @E
F       INC  R1             * GET SECOND PARAMETER
        BLWP @NUMREF
        BLWP @XMLLNK
        DATA >1200
        MOV  @>834A,R4
        CI   R4,>0000       * CHECK IF VALID VALUE
        JGT  G
        B    @E
G       CI   R4,>0008
        JLT  H
        B    @E
H       INC  R1             * GET THIRD PARAMETER
        BLWP @NUMREF
        BLWP @XMLLNK
        DATA >1200
        MOV  @>834A,R5
        CI   R5,>0000       * CHECK IF VALID VALUE
        JGT  J
        B    *R11
J       CI   R5,>00A1
        JLT  K
        B    @E
*  DEFINE CHARACTERS
*
K       LI   R2,>0008       * HOW MANY BYTES TO WRITE & INCR FOR R0
        LI   R0,>0640       * VDP WRITE ADDRESS
B       LI   R1,D1          * ADDRESS OF CHARACTER DATA IN R1
A       BLWP @VMBW          * WRITE 8 BYTES INTO CHAR TABLE
        A    R2,R0          * INCREMENT BY 8 FOR NEXT CHAR
        INC  R1             * SHIFT 1 PIXEL ROW DOWN DATA
        CI   R1,D1+8        * LAST DATA ADDRESS ?
```

```
       JLT  A              * NO KEEP GOING
       CI   RØ,>Ø8ØØ       * LAST CHAR TO DEFINE ?
       JLT  B              * NO DO NEXT CHAR
*
*   DEFINE CHARSET COLOURS
*
       LI   RØ,>Ø319       * VDP ADDRESS FOR CHARSET 1Ø
       LI   R1,D2          * SET COLOUR
       LI   R2,>ØØØ7       * 7 BYTES TO WRITE
       BLWP @VMBW          *
*
*   DETERMINE CHARACTER TO USE
*
       SLA  R4,3           * MULTIPLY BY 8
       AI   R4,>ØØCØ       * 192 LESS OFFSET OF 96 = 96+8 = 1Ø4
*                          * WHICH IS 1ST CHAR OF CHARSET 1Ø
       SWPB R4             * CHAR TO USE
       LI   R6,>ØØ2Ø       * ROW DECREMENT VALUE
       LI   RØ,>Ø261       * SET ROW 2Ø COL 2
       A    R3,RØ          * COLUMN TO USE
*
*   DRAW BAR ONE PIXEL ROW AT A TIME
*
N      CLR  R7             * RESET COUNTER
       MOV  R4,R1          * PREPARE TO WRITE 1ST BYTE
M      BLWP @VSBW          * WRITE ONE BYTE
       DEC  R5             * REDUCE BAR BY ONE PIXEL ROW
P      CI   R5,>ØØØØ       * FINISHED ?
       JGT  L              * NO. CONTINUE
       B    *R11           * YES. RETURN TO BASIC
L      AI   R1,>Ø1ØØ       * GET NEXT CHARACTER
       INC  R7             * INCREMENT COUNTER
       CI   R7,>ØØØ8       * WRITTEN ONE FULL CHARACTER ?
       JLT  M              * NO. CONTINUE
       S    R6,RØ          * YES. DECREMENT ROW
       JMP  N              * CONTINUE NEXT ROW
*
*   ERROR IN PARAMETER
*
E      LI   RØ,>13ØØ       * BAD VALUE ERROR
       BLWP @ERR
            END
```

```
100 REM   BAR GRAPH DEMONSTRATION
110 REM  TI BASIC W/MINI-MEMORY OR
120 REM  EDITOR/ASSEMBLER MODULE
130 REM   BY PHIL WEST AND BERNIE ELSNER
140 CALL CLEAR
150 RANDOMIZE
160 PRINT "PLACE FILE 'BGRAPH/O' IN"
170 PRINT "DISK DRIVE ONE THEN":"PRESS ENTER"
180 INPUT E$
190 PRINT "LOADING MACHINE LANGUAGE":"PLEASE WAIT"
200 CALL INIT
210 CALL LOAD(28706,0,0,0,0,0,0,0,0)
220 CALL LOAD("DSK1.BGRAPH/O")
230 PRINT :"WHICH MODULE ARE YOU USING?":"ENTER"
240 PRINT "  E - FOR EDITOR/ASSEMBLER":"  M - FOR MINI-MEMORY"
250 INPUT E$
260 IF E$="M" THEN 350
270 IF E$<>"E" THEN 250
280 PRINT :"PLACE EDITOR/ASSEMBLER"
290 PRINT "DISK A IN DRIVE ONE THEN":"PRESS ENTER"
300 INPUT E$
310 CALL CLEAR
320 CALL LOAD("DSK1.BSCSUP")
330 REM   RESERVE SPACE FOR CHARACTER DEFINITION
340 REM   IN MACHINE LANGUAGE ROUTINE
350 FOR I=104 TO 159
360 CALL CHAR(I,"")
370 NEXT I
380 CALL CLEAR
390 CALL SCREEN(15)
400 PRINT "MICRO RETAIL PROFITS 1982-84"
410 PRINT "    (Q)UIT OR (R)EPEAT"
420 F=1
430 FOR I=1 TO 28
440 F=F*1.195
450 CALL LINK("BGRAPH",I,1,F)
460 NEXT I
470 FOR D=1 TO 800
480 NEXT D
490 FOR I=1 TO 28
500 CALL VCHAR(1,I+2,32,20)
510 CALL LINK("BGRAPH",I,6,F)
520 F=F/1.195
530 NEXT I
540 FOR D=1 TO 800
550 NEXT D
560 CALL HCHAR(1,1,32,640)
570 REM  GENERATE RANDOM PARAMETERS
580 FOR I=1 TO 28
```

```
590 B=INT(RND*7+1)
600 C=INT(RND*160+1)
610 REM  DRAW BAR WITH M/L ROUTINE
620 CALL LINK("BGRAPH",I,B,C)
630 NEXT I
640 CALL KEY(0,K,S)
650 IF K=82 THEN 380
660 IF K<>81 THEN 640
670 END
```

EDITOR/ASSEMBLER MANUAL REFERENCES

The following references will provide you with some more information on mixing assembly with BASIC.

Read these sections:

Section 17.1 page 273 through Section 17.2.6 page 289.
Section 18.2.5 page 300.
Section 21.1 page 326.
Section 24.4 page 410 through Section 24.4.9 page 418.
Section 24.11 page 440 through Section 24.11.3 page 442

Look up these terms in the glossary:

GPL
Loader
Utilities
Workspace
Workspace Pointer Register

PARTING WORDS

There are several points aside from the actual understanding of TMS9900 assembly language that can help make programming in this language somewhat easier and more productive. These involve certain work habits and procedures and appreciating the task at hand when creating assembly programs.

No matter how proficient you are now or may become with assembly language programming, it is almost never practical or sensible to simply begin typing lines of code for a new program. Though this may be possible with TI BASIC, it is not advisable with assembly. Once you have decided on an application you would like to write in TMS9900 assembly language, take the time to sketch out your program with pencil and paper before going to the computer. As stressed earlier in this book, assembly language requires a great deal of detailed consideration for each and every little thing you want to do.

Consult any reference materials you have at your disposal to insure that the first draft of your program is as free of errors in logic and syntax as possible. Make use of sections of programs you have written previously. Many professional programmers maintain an inventory of subroutines they have developed that can be reassembled modularly to form the basis of a new application program.

At a very fundamental level, you should pay close attention to the structure and design of your programs. Assembly languages are basically free-form languages. As long as the rules of syntax are obeyed, the directives, instructions, and data can appear in any order. This is one case where freedom is not necessarily a good thing. The program examples given in this book all follow a prescibed pattern or form. Directives, data statements, references, and equates are all listed first. These are then followed by the actual program instructions. As much as possible, your programs should incorporate a "top down" design. As you read the program from top to bottom, you should be following the logical sequence of events as they occur.

If you were to create a painting, a symphony or a skyscraper, careful design and planning would be crucial to the successful implemention of your creation. The same applies to creating an assembly program. You need to consider in advance how you will tackle the task at hand. You must anticipate the needs of the program itself, such as temporary storage fields for calculations and data manipulation. Look for redundant functions that could be coded as subroutines. Document your programs with comments and labels that explain the program logic. Consistently align the labels, instructions, and operands on the column boundaries they will occupy. This makes your program easier to read. Paying attention to neatness can usually make errors easier to detect.

Develop your own style and structure in designing programs and stick to it. Borrow freely from other examples you find to substitute for, or improve upon, your own designs. You can learn a great deal from other assembly programmers. A good place to find this kind of help is at a local users group. And, remember that one of your best teachers is simple trial and error. If you have been wondering if something will work, try it.

Several products are available that can help in program development and debugging. As mentioned, these include special hexadecimal calculators and software packages such as the TI Programming Aids series. There are also disassemblers that can convert object code back into source code. Both the Line-by-Line assembler and the Editor/Assembler come with interactive debugging programs to assist you in your programming. Pros and hobbyists alike benefit from these kinds of productivity tools. However, it should be stressed that none of these is a substitute for a fundamental understanding of assembly language principles.

Remember to make back up copies of programs and data files before running the programs or accessing the data. This can save you headaches and frustration. Good computer work habits become more critical when working in assembly language.

If you are using Editor/Assembler, a printer is one of the most useful peripheral devices you could own. The listings you can produce with it are invaluable for debugging programs and creating documentation. It will help to have a printed listing to do a "walk through." This involves reading through the program a line at a time. As you encounter each line of code, make note of what is stated and record the results as you understand them on a piece of paper. For example, draw a set of columns and label the top of each with the name of the label or register involved. Then, as values are initialized or changed, record these values in the correct column. Draw a line through the previous value when a value changes. Do not erase it since you may need to refer back to it. Make note of the line number that caused the change. This will show you the contents of each register or field at each step. If you encounter an instruction to add the contents of two registers having values that were determined 20 lines before, the values to be added will be in the corresponding columns you created.

Developing good work habits and disciplines and learning to pay attention to details will assist you in assembly language programming and in all other types of interaction with your computer. Practice makes perfect. The more you work with assembly language, the better you will become at it. Of course, this will require some effort and determination on your part. You will be rewarded for your effort with an increased understanding of your computer and a powerful tool that puts its potential at your command—TMS9900 Assembly Language.