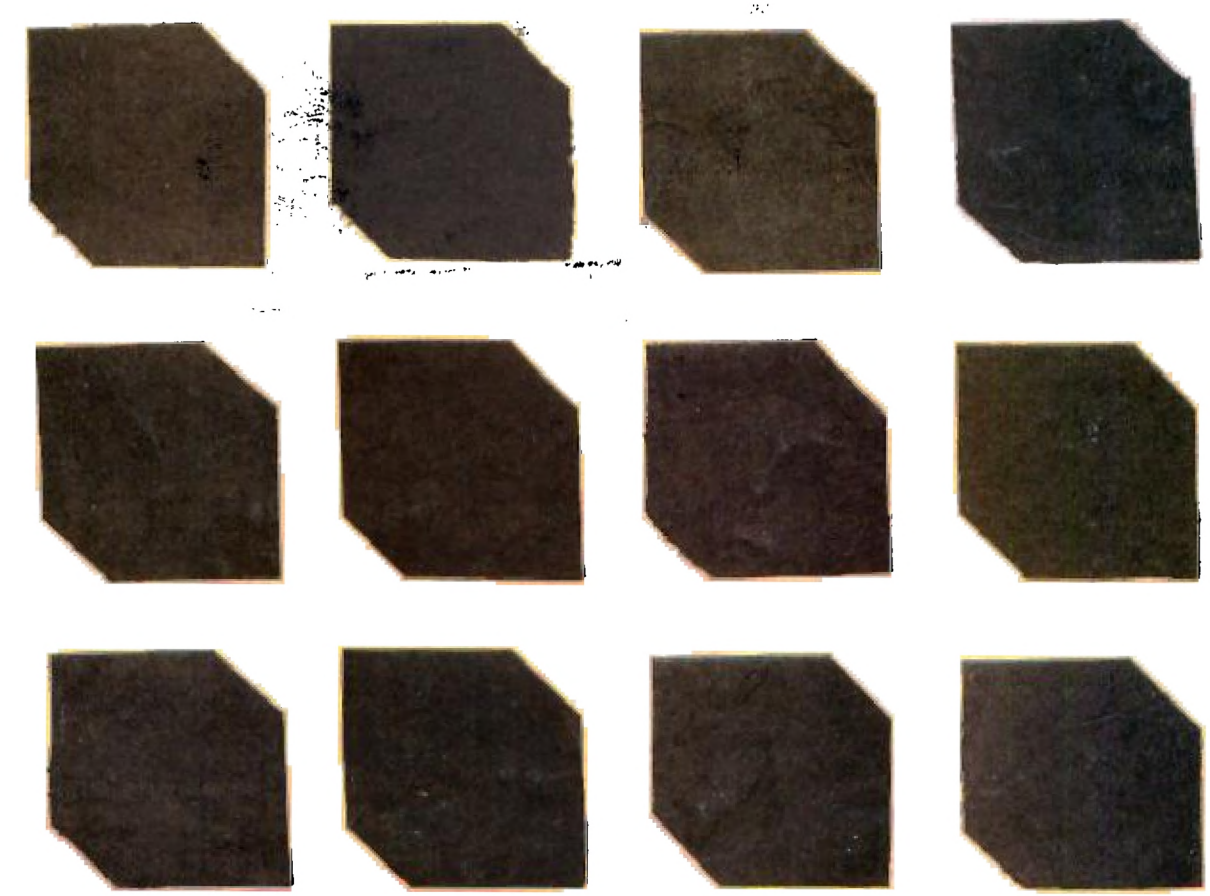
BEGINNING ASSEMBLY LANGUAGE FOR THE TI HOME COMPUTER



BEGINNING ASSEMBLY LANGUAGE FOR THE TI HOME COMPUTER

For correspondence regarding this book
address the following:

D & D PUBLISHING Co.
3177 Bellevue
Toledo, Ohio 43606

Printed in the United States of America 1984

1

INTRODUCTION

There is something really big underlying the BASIC language!

Many gifted programmers have considered writing exciting games for the TI Home Computer only to be faced with the limitations of the cumbersome BASIC language. It does not take one long to realize that it is simply not possible to accomplish all that arcade style games entail using BASIC alone. BASIC is sometimes just too slow.

There is essentially nothing wrong with using BASIC if you're programming operations don't require a great deal of speed. But if you are writing programs which have a lot of things happening simultaneously, such as a number of objects flying around the screen with the program trying to keep track of coincidence checks, BASIC just can't do the job.

BASIC by its very nature tends to use up a lot of memory in a short period of time. For these reasons and the ones previously alluded to, you may want to consider adding program modules written in assembly language to your BASIC programs. Or even writing your complete program entirely in assembly language.

This book is designed to help the beginner in introducing him or her to assembly language. The book assumes that you have no previous experience in programming other then BASIC. If you already know BASIC, that is fine. If you are already developing programs in assembly language, that is even better.

This book was designed as a study text. That is, it was meant to be read cover to cover, each chapter building on what was learned in the preceding chapters. If something is discussed that you do not quite understand after a thorough reading, go on as it will probably become clear in later sections. Take the time to complete the study questions at the end of each chapter. They will reinforce important concepts.

This book begins with the fundamentals. Chapter 2 covers the binary and hexadecimal numbering systems. It also discusses important terms and concepts that will be carried throughout the book. Make sure you completely understand chapter 2 before proceeding.

CONTENTS OF THIS BOOK

This book contains 14 chapters. In chapter 2 you are introduced to the counting system that the computer uses to keep track of numbers. You are also introduced to the **hexadecimal system** which greatly simplifies programming.

Chapter 3 discusses the assembler, memory utilization and the internal registers of your Home Computer. It also explains how assembly language programs are developed and written. Additionally, you are introduced to the **source statement**, which is a programing line in assembly language akin to a BASIC statement.

Chapter 4 introduces the instruction set. The first topic taken for discussion is **Addressing Modes**, or ways to inform the computer exactly where data or information can be found in memory. Subsequent sections of this chapter introduce you to the **Instruction Set** with each instruction discussed at length as to its usage and purpose. Numerous examples are used to dramatize important points.

In Chapter 5 you learn about **Assembler Directives**. These consist of instructions to the assembler program that can significantly reduce program development time on your part.

Chapter 6 discusses **Utility programs** in-depth. These are already constructed assembly language programs that are available to you. Again, numerous examples are provided to illustrate important points.

Chapters 7, 8 and 9 discuss screen **Graphics**, **Sprites** and **Sound** control. You learn how to control complex screen graphics as well as how to incorporate sound into your programs.

Prior to chapter 10 this book discusses how to create assembly language programs using the Editor/Assembler package. Chapter 10 is a complete description of how to create assembly language programs using the line-by-line assembler and the Mini-Memory module. Explicit instructions are given explaining the differences and how to create programs that will run with either system configuration.

Chapter 11 outlines the conversion of many BASIC commands into their assembly language equivalents. This is done to illustrate

general assembly language concepts.

Chapter 12 outlines BASIC support routines that are available. It explains how to link BASIC programs with assembly language programs. It also outlines how parameters are passed between the two types of programs.

Chapter 13 presents a brief description of the advanced mathematical routines that are available. Linking to console resident routines is also discussed.

This book provides four appendices for your convenience. Appendix A contains tables that aids in interchanging decimal and hexadecimal numbers. Appendix B outlines the TMS9900 Instruction Set. Appendix C lists the Assembler Directive set. Appendix D is perhaps the most interesting, it provides some source code for frequently used assembly language game modules. You can operate joysticks, simulate gravity, scroll the screen, create delays ect...

GOOD LUCK!

CONTENTS

2

HOW COMPUTERS COUNT

The difficulties encountered in learning assembly language have often been greatly exaggerated. In fact, once the instructions and the rules that govern them are understood, programming in assembly language becomes almost as easy as programming in BASIC.

All humans are born with ten fingers and toes and hence it was natural that our mathematics would develop along the base ten numbering system. However, there is no natural "law" that states this must be so. A computer is designed along a base 2 or **binary numbering system**. It is made up of only two digits, 0 and 1 (in contrast to the decimal system which is made up of the digits 0 through 9). When you are working with the binary numbering numbering system you are talking in the computers own language. The computer can act directly upon instructions rather then having to go through an interpreter first as is necessary with any higher level language like BASIC.

There is one additional numbering system that you should become familiar with in this chapter. This is a base sixteen or **hexadecimal numbering system** or simple **HEX**. The HEX system is made up of the digits 0 through 9 and letters A through F. When programming in assembly language the computer assumes all numbers that you enter are decimal numbers unless you precede the number with a "greater than" symbol (>). The greater than symbol indicates to the computer that the number following it is in hexadecimal notation.

124 (Decimal) **>7C (HEX)**

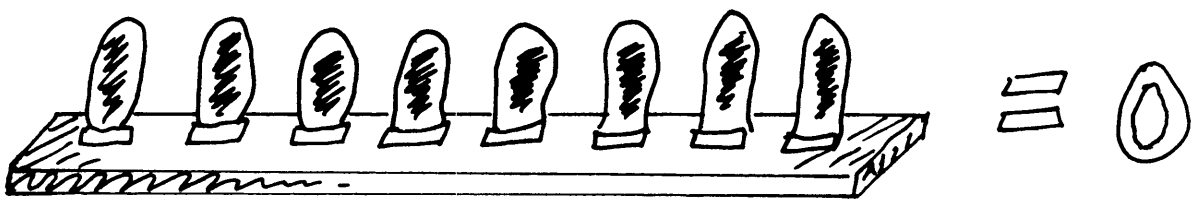
This chapter is a basic introduction to computer numbering systems. It is aimed at those who have no or limited knowledge in this area. If you already understand these concepts and how they apply to assembly language programming, feel free jump ahead to the next chapter.

2.0 BINARY NUMBERS

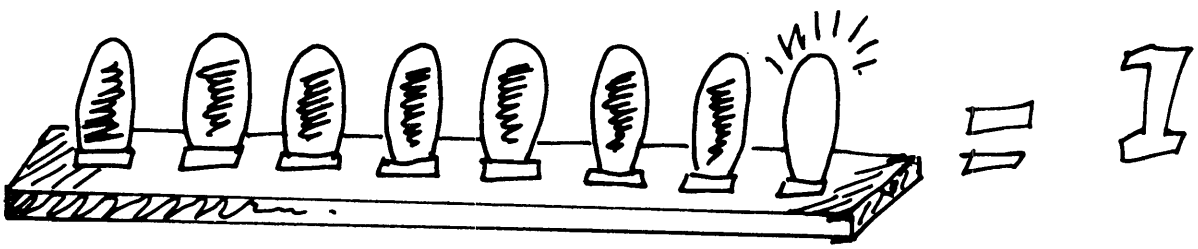
The computer stores all the information contained within it in an area called the **memory.** Memory can be thought of as a large collection of electrical switches. Each switch can be either "on" or "off" and each can be set or reset by the computer as needed. Each individual switch can be thought of as the computers smallest single memory cell. This single memory cell is known as a **BIT** which is short for Binary DIgiT. A bit holds the smallest piece of information that the computer can handle. A bit is either on or off, true or false, plus or minus. It has no in-between states.

The On and Off settings of the memory bits correspond to the two digits that make up the binary numbering system. The binary system consists of the two digits 0 and 1 and is the fundamental system the computer uses to keep track of numbers. The digits are represented by 0(Off) and 1(On).

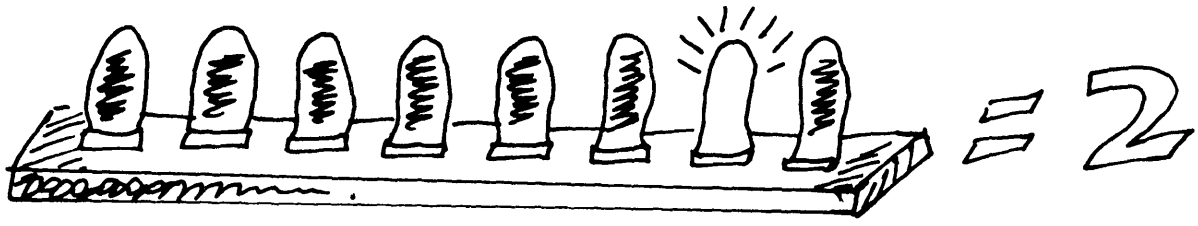
In your Home Computer groups of **eight bits** are lumped together to form a single **byte.** It might be easier if you think of a byte as a row eight lightbulbs mounted on a long board. Each lightbulb represents a single bit and can be either on or off. The entire board with its eight lightbulbs is taken as one byte. In the following sections we will see how the computer can use these bits and bytes to store information.



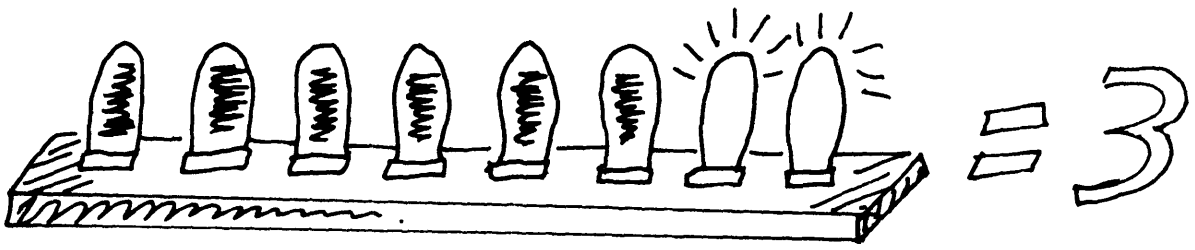
Looking at the above illustration of our byte we see that each of the lights (bits) are currently turned off. From this we can say that the byte is representing zero value. In computer language it is said to be "holding" a zero. Now consider that we want this byte to represent the number one instead of zero. As we watch the light (bit) on the far right comes on:



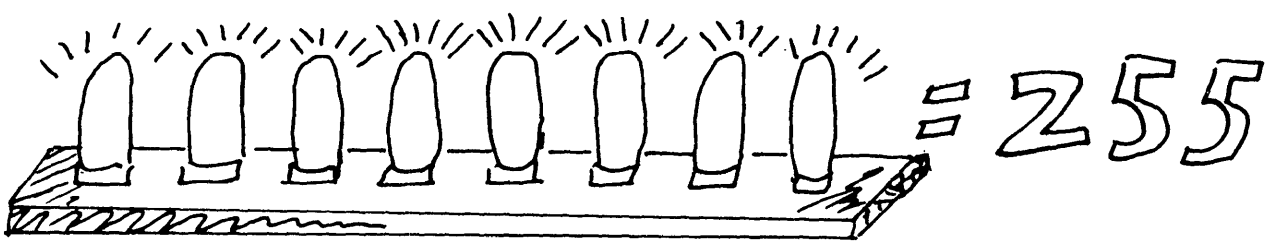
The column on the far right of our byte is the one's column and hence the byte on the preceding page would represent or "hold" a value of one. If we wanted our byte to hold a value of two instead we would turn on the next bit in the row like so:



And to represent the number three we simple add the values of the last two bits together like so:



By simply looking at a byte, checking to see which bits are turned on, and adding their values together the computer can tell the value of the number being held there. Each bit has its own special position on the byte. Starting on the right and proceeding to the left, each bit is worth twice what the one before it was. Another way to think about it is to consider each bit (from right to left) as an increasing power of two. Thus the rightmost bit is 2 to the power of 0 or 1, the next bit is 2 to the power of 1 or 2, then next 2 to the power of 2 or 4, and so on until the leftmost bit is reached which is 2 to the power 7 or 128. By adding combinations of bits that are turned on together the value of any number from 0 (all bits off) through 255 (all bits on) can be represented:



Lets review, eight bits together make up a single byte. A single byte can hold any value ranging from 0 to 255 decimal. The following examples are binary (byte) representations of some decimal numbers. Keep in mind that each 1 or 0 represents a bit that is either ON(1) or OFF(2). The bits are divided into two groups of four bits each to make them easier to read:

BINARY	DECIMAL
0010 0010	
(32)+(2)=	34
0100 0010	
(64)+(2)=	66

Normally you would not have to add binary numbers together when programming, this function being performed by the computer. However, to provide a complete presentation we will briefly discuss the addition of binary numbers.

When adding binary numbers together you follow essentially the same procedure as when adding two decimal numbers together. For example, when adding the values 6 and 8 together you must carry a 1 into the "tens" column in order to arrive at the correct result of "14". Similarly, when the two binary digits 1 and 1 are added together, a 1 must be carried into the two's column. Thus the addition of 0000 0001 with 0000 0000 becomes 0000 0001 and the addition of 0000 0001 with 0000 0001 becomes 0000 0010. The following illustrate some further examples of binary addition:

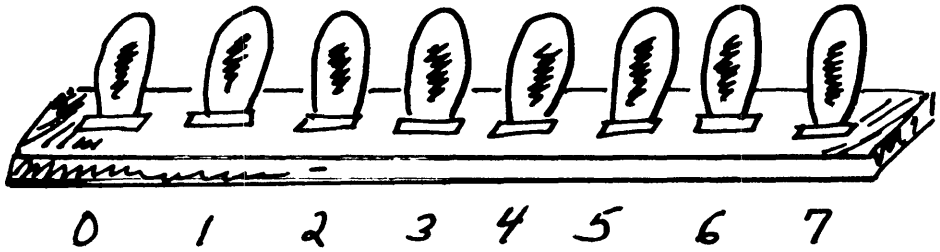
```
   *          **          **    *        *carried 1's
   1          11          11    1
  0101       0111        0110 0110
+ 0001     + 0110      + 0111 0011
  0110       1101        1101 1001
```

The first problem involves a carry of one from the first column to the second (1+1). This carries over to the second column which contains only two 0's. Adding the carried 1 makes the result under this column a "1".

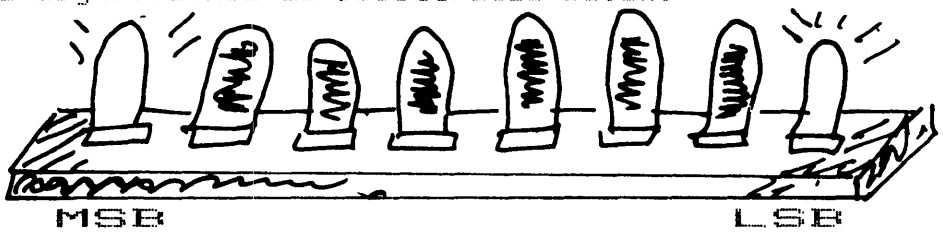
2.1 SIGNED NUMBERS

Up to this point we have been discussing how to represent positive numbers with the binary system (using bytes). To bits and bytes we must return to our row of eight bits that we discussed in previous sections. Remember that each bit represented

a certain value that was determined by its row position on the byte. To make them easier to refer to, bits are numbered 0 through 7 starting on the left and proceeding to the right (in contrast to their value which increases from right to left). The numbering of bits is illustrated below:



Bits are also said to become more **significant** as they increase in value. That is, bit 7 is considered the **least significant bit (LSB)**, and bit 0 is the **most significant bit (MSB)**. Also, bit 0 is more significant than bit 1 and bit 1 is more significant than bit 2 and so on down the line. Significance than is tied to the relative value of a bit. As the relative value increases, so does the bits significance as illustrated below:



When a byte holds a signed number, only the 7 least significant bits hold the value of the number (bits 1 thru 7). The most significant bit (bit 0) is reserved and is used to indicate the sign of the number being held. If this bit is set to "1" then it indicates that the number being held is a negative number. If this bit is reset to 0 then it indicates that the number being held is a positive number.

As you may have already guessed, a byte that holds a signed number uses bit 0 to hold the sign. Therefore it can't hold as wide a range of values. Bytes holding positive numbers can only hold values ranging from 0 (binary 0000 0000) to 127 (binary 0111 1111) while bytes holding negative numbers can hold values ranging from -1 (binary 1111 1111) to -128 (binary 1000 0000).

You may be wondering why -1 is represented in binary as 1111 1111 instead of 1000 0000. The reason for this is that negatively signed numbers are represented in what is known as their **2's compliment form.** By using 2's compliment to represent negative numbers the dilemma of having zero be represented by all 0's (positive zero) and all 0's with a 1 in the sign position (negative zero) are avoided.

To find the binary representation of a negative number (that is, to find its two's compliment form) simply reverse each bit, that is change each 1 to 0 and each 0 to 1, then add 1 to the result. The following example illustrates how to find the 2's compliment representation of -65:

```
    0100 0001      +65

    1011 1110      Reverse all bits.
+   _________1     Add one.
    1011 1111      -65
```

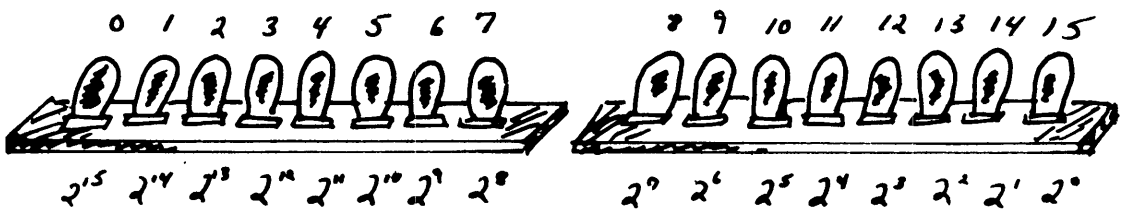
The reverse procedure (reverse all bits and add 1) can also be used to find the positive form of a negative number.

2.2 COMPUTER WORDS

A bit is the smallest piece of information that the computer can hold. The computer lumps 8 of these bits together to form a single byte which it can use to store usable information. By now you should begin to see some limitations with this system. For example, using bytes alone you could only represent unsigned numbers whose values range from 0 to 255 or signed numbers whose values range from -128 to +127. To represent numbers larger than this we must devise some alternate scheme. The simplest approach would be to hook two bytes together in order to form a larger number of bits from which to draw information.

Two bytes hooked together in this fashion are referred to as a single WORD. The left byte contains the first 8 bits that make up the left-half of the "word" while the right byte contains the second group of 8 bits that form the right-half of the "word". The bits are numbered consecutively left to right from bit 0, the left-most bit on the left byte, through bit 15 which is the right-most bit of the right byte. The value of each bit is double as we move from right to left along the bits. For example:

BIT NUMBER:



VALUE OF BIT

Notice that by linking two bytes together in this manner to form a single word we can now represent a much greater range of

numbers.

To sum up, in your Home Computer most chunks of information are processed in units referred to as words. Each word is made up of two bytes. Each byte is made up of eight bits.

For words that contain signed numbers, bit 0 (the left-most bit of the left byte) is used to hold the sign of the number. Words can hold signed values that range from 0 (0000 0000 0000 0000) to 32,767 (0111 1111 1111 1111). Words holding negative numbers can hold values ranging from -1 (1111 1111 1111 1111) through -32,768 (1000 0000 0000 0000). Keep in mind that negative numbers are represented in their two's compliment form. The following is a graphic representation of -4356:

2.3 HEXADECIMAL NOTATION

When computers were in their infancy programmers had to enter each byte of binary code by hand. Not only was this a very tedious and time consuming process, but it was extremely prone to error as well. For example, a binary number like 0000 1110 could easily be transposed into the entirely new value 0000 1101.

The HEX system (short for **hexadecimal**) was designed to speed up the process of writing in binary code. The following chart compares the Decimal, HEX, and Binary numbering systems:

DECIMAL	HEX	BINARY
0	>00	0000 0000
1	>01	0000 0001#
2	>02	0000 0010
3	>03	0000 0011
4	>04	0000 0100
5	>05	0000 0101
6	>06	0000 0110
7	>07	0000 0111
8	>08	0000 1000
9	>09	0000 1001
10#	>0A	0000 1010
11	>0B	0000 1011
12	>0C	0000 1100
13	>0D	0000 1101
14	>0E	0000 1110
15	>0F#	0000 1111

Note that (#) signifies that the digits begin to repeat on the preceding page (10's decimal, 16's HEX, 2's binary).

If you study these systems you find that in decimal you begin 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, then start again in the 10's column: 10, 11, 12, 13, 14, ... and so on.

With HEX you count >0, >1, >2, >3, >4, >5, >6, >7, >8, >9, >A, >B, >C, >D, >E, >F, then start again in the 16's column: >10, >12, >13, >14, >15, ... >18, >19, >1A, >1B, >1C, ... and so on.

In both decimal and hexadecimal numbering systems the individual digits have some "weight" which is a power of the base. In the HEX system the base is sixteen so each digit has a value 16 times the value of the digit to its immediate right (as opposed to the decimal system where each digit has a value 10 times the value of the digit to its immediate right). For example, the hexadecimal number >4CEF has a decimal value of 19,695 because:

$$(4\times16^{3})+(C\times16^{2})+(E\times16^{1})+(F\times16^{0}) = 19,695$$

reduces to the decimal form:

$$(4\times4096)+(12\times256)+(14\times16)+(15) = 19,695$$

where C=12, E=14, and F=15 from the table on page 11.

When writing in assembly language all HEX numbers are designated with a "greater than" sign (>) in front of them to differentiate them from decimal values. The following are some HEX equivalents of decimal values:

	UNSIGNED NUMBERS	
HEX	DECIMAL	BINARY
>0A	10	0000 0000 0000 0000
>AA	170	0000 0000 1010 1010
>B3F	2,879	0000 1011 0011 1111
>FFFF	65,535	1111 1111 1111 1111
>FE03	65,283	1111 1110 0000 0011
>0214	532	0000 0010 0001 0100
	SIGNED NUMBERS	
>FC	-4	1111 1111 1111 1100
>E9	-23	1111 1111 1110 1001
>08	8	0000 0000 0000 1000
>7FFF	32,767	0111 1111 1111 1111

Learning to work with hexadecimal numbers is perhaps the biggest hurdle to get over when trying to master assembly language. You should not be disillusioned if everything is not

crystal clear up to now after all, this counting system is unnatural. You should spend some time now practicing the exercise at the end of this chapter. You should at least be fluent in converting hexadecimal numbers into their decimal equivalents and vice-versa before proceeding even if you don't quite understand what is going on yet.

To sum up, in order to figure out the decimal value of a HEX number, simply multiply the second digit by 16, the third by 16 squared the fourth by 16 cubed and add all four values together. Thus >12A becomes (1x256)+(2x16)+(10x1)=298.

HEX at first does seem impossibly confusing. Do not let this discourage you as the system will probably become second nature to you after you have worked with it for awhile. You can quickly look up HEX values that you need in a hurry in Appendix A at the end of this book. Remember, all HEX numbers are distinguished by placing a "greater than" sign (>) in front of them: >0A or >1222.

CHAPTER 2 STUDY EXERCISES

1. Convert the following decimal values to their binary equivalents:

   (A) 15 (B) 24 (C) 30,121 (D) -10,250

2. Convert the following unsigned binary values to decimal:

   (A) 0100 (B) 0010 0100 1110 1101 (C) 1001 0000 0000 0000

3. Write all four numbers in exercise 1 in hexadecimal notation.

4. List the decimal equivalent of >1C34 if:

   (A) The value represents a signed number.

   (B) The value represents a unsigned number.

3

THE ASSEMBLER

In the last chapter we learned that the computer speaks in a binary code. We also learned that binary code is the most efficient and fastest executing language. In addition, we learned an alternate method of designating numbers; that being the hexadecimal system.

Early on programmers found it difficult to program instructions into the computer using binary codes. For instance, to enter the instruction that would add two numbers together required having to type in the binary code 1010 0000 0000 0000, or the HEX equivalent, >A000. Likewise, to enter the subtraction instruction required having to enter the binary code 0110 0000 0000 0000, or the HEX equivalent, >6000. As can be easily seen, this is not only a time consuming process, but is extremely prone to error as well.

Eventually someone got the idea to replace the binary commands with english abbreviations that programmers could easily remember. In this way an addition instruction could be typed in as "A" instead of 1010 0000 0000 0000, and a subtraction instruction could be written as "S" instead of 0110 0000 0000 0000. A separate program referred to as the "**assembler**" is then used to convert these abbreviations into their binary equivalents.

When a program is first written in this "assembly language" it cannot be run on the computer yet since the computer does not understand the abbreviations. Before a program can be run it must be **assembled** by the **assembler program**. There are thus two versions of an assembly language program. The first version written by you using the abbreviations is termed the **source program** (or **source code**) while the second binary version created by the assembler program is termed the **object program** (or **object code**).

In summary, the purpose of the assemble program is to convert the source code which you have written into object code which the computer can understand.

3.0 REGISTERS

Before we advance too far into assemble language programming proper, it would be useful for us to discuss how the computer keeps track of instructions and how it follows through with them in a neat, orderly manner. The electronic brain of your computer is the TMS 9900 processor. It has the capability to perform a wide variety of tasks quickly and efficiently.

If we could look down into the computer we would be able to see distinct areas that serve specific functions. One area is called **RAM** which stands for **Random Access Memory.** RAM contains a large number of free bytes. You can, as the name implies, randomly access any of the bytes located here. This is the area where your program instructions are stored when you type them into the computer. Thus, RAM can be though of as a blank slate waiting for you to type in information.

Another area is referred to as **ROM** which stands for **Read Only Memory.** This is an area where the computer permanently stores a set of instructions that it can refer to when needed. For instance, when you type in a BASIC command, ROM is where the instructions that translate the BASIC command into binary code reside.

The third major area of the computer is termed the **CPU** or **Central Processing Unit.** It is the heart and soul of the computer. The CPU continuously takes in numbers from memory locations all over the computer. These numbers can then be sent out unchanged to other locations, or they can be compared, added to, or otherwise modified before being sent back to RAM or ROM. The CPU can perform all these tasks with the help of some special "tools". These tools are referred to as **Registers.** A Register can be thought of as a memory word that is reserved for a specific purpose (remember, a word is made up of 2 bytes hooked together). Registers located in RAM that you can alter during programming are referred to as **Software Registers.** Registers located in ROM that can be used only by the CPU are termed **Hardware Registers.** A set of sixteen consecutive Registers is referred to as a **Workspace.**

It may be helpful to think of a Register as an area of memory where you can store information that you want the CPU to perform some operation on. For example, suppose you wanted to add two numbers together. You would first place the values to be added in two Registers and then instruct the computer to add them together and place the sum into a third Register. Registers can be located anywhere in RAM as long as you tell the computer where they are. In later chapters we will discuss how this is done.

In your Home Computer you have a total of sixteen Software Registers (termed a workspace) available to you. Each Register is

one word (2 bytes) in size. These sixteen Registers are numbered R0 through R15. These sixteen Registers are collectively referred to as your **Workspace Registers.**

In addition to the Software Registers available to you there are three Hardware Registers that are used by the CPU to keep track of things. These are as follows:

1. PROGRAM COUNTER REGISTER
2. WORKSPACE POINTER REGISTER
3. STATUS REGISTER

The following sections describe the three Hardware Registers in great detail.

PROGRAM COUNTER REGISTER (PC)

The Program Counter Register (PC) keeps track of the location of the next instruction to be executed by the CPU when it is running a program. In this way a sequential and orderly flow of instructions is maintained.

WORKSPACE POINTER REGISTER (WP)

The Workspace Pointer Register (WP) keeps track of the location in memory of the current Software Workspace. This is the pointer that informs the computer where your Software Workspace area begins in RAM.

Each byte in RAM is numbered so that the computer can find it. This number is referred to as the **Address** of the byte. This is similar to how the location of each house in a large city is designated by its street address. With this in mind it can be stated that the Workspace Pointer Register holds the beginning address of the current Software Workspace.

STATUS REGISTER (ST)

The Status Register is important in that it reports to the CPU about the current **Status** of things. For example, when the computer compares two numbers together it is useful to record the result of this comparison somewhere in memory. That is the purpose of the Status Register; it "holds" the information long enough for the CPU to make a decision based on it. Remembering that a Register is made up of sixteen bits, the Status Register reports various status conditions in the first six of its bits (0-5). The four least significant bits (12-15) hold information important towards interrupting the computer; but we will have more on interrupts

later. Bits 7 through 11 are not used by the Status Register.

Each bit in the Status Register can be thought of as a **flag** that signals some piece of information to the CPU. Every time a bit is set to 1, it signals to the CPU which may act on the flag, or ignore it depending on your program instructions.

The following figure demonstrates how the "flags" are arranged in the Status register:

L>	A>	EQ	C	OF	OP	X	NOT----USED					INTERRUPT MASK			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

BIT NUMBER

L> -- LOGICAL GREATER THAN BIT
A> -- ARITHMETIC GREATER THAN
EQ -- EQUAL BIT
X -- EXTENDED OPERATION
C -- CARRY BIT
OF -- OVERFLOW BIT
OP -- ODD PARITY BIT

The Status flags signify the following conditions:

BIT 0: LOGICAL GREATER THAN (L>), is set to 1 if a larger unsigned number is compared to a smaller unsigned number.

BIT 1: ARITHMETIC GREATER THAN (A>), is set to 1 if a larger signed number is compared with a smaller signed number.

As we have noted in the preceding chapter, the most significant bit (bit 0) of a word holds the sign of the number (0 for positive, 1 for negative). For positive numbers, the remaining bits represent the binary value of the number. For negative numbers, the remaining bits represent the two's compliment form of the binary number.

BIT 2: EQUAL (EQ), is set to 1 when two numbers being compared are equal. The equal bit is set regardless if the comparison is between two signed numbers or two unsigned numbers.

BIT 3: CARRY (C), is set to 1 if an add operation produces a carry or if a subtraction operation produces a borrow of bit 0; otherwise it is reset to 0. The Carry bit also holds the value of a bit that has been rotated or shifted out of a Register or Memory location.

BIT 4: OVERFLOW (OF), is mainly an error indicator. It is set to 1 when the addition of two like signed numbers, or the subtraction of two oppositely signed numbers, has produced a result that is too large or small to be represented correctly by a single word.

Additionally, the OF bit is set to 1 if, during an arithmetic left shift, the most significant bit of the Register being shifted

changes value.

Also, during divide operations the OF bit is set to 1 if the most significant 16 bits of the dividend are greater than or equal to the divisor.

BIT 5: ODD PARITY (OP), is set to 1 when the parity of the result of a byte operation is odd. The OP is reset to 0 when the parity of the result is even.

The parity of a byte is said to be odd when the number of bits contained within it having a value of 1 is odd. For example the byte 0001 1111 is said to have odd parity because it has an odd (5) number of bits set to 1. Even parity is just the opposite.

BIT 6: EXTENDED OPERATION (X), is set to 1 when software implemented extended operation is initiated. However, the instruction XOP (for extended operation) is not available on all Home Computers. The only way to see if your computer supports this instruction is to try it.

BITS 7-11: UNUSED

BITS 12-15: INTERRUPT MASK, allows the TMS 9900 to recognize interrupt requests from peripheral devices hooked into the system. If the peripheral device has a level number less than or equal to the value in the interrupt mask, it is permitted by the CPU to interrupt a running program. Thus, if the four bits making up the interrupt mask are set at 2 (0010), then any device with a level 0, 1, or 2 may interrupt a running program. In your Home Computer the interrupt mask is always set at 2 (0010). Because of this only values of 2 and 0 are useful.

Everybody has interruptions in their lives. Some of these are necessary such as when a newborn cries for food, you must stop what you are doing attend to the infants needs. While other times you may be to busy to be interrupted, such as when the phone rings during your favorite T.V. show; you may choose to let it go unanswered! The same is true for the computer. Sometimes peripheral equipment needs information from a running program and interruptions are the only way they can get it. Also, some ROM routines such as automatic sprite motion or sound generating routines need to be able to interrupt your running program in order to execute.

When you first turn on the computer all the Status bits are reset to 0. Don't worry if your not quite sure yet as to the significance of the Status Register, it should become clearer as we progress.

3.1 WRITING PROGRAMS

When first putting a program together from scratch you should follow a certain logical sequence of steps. These steps are summed up below:

1. Decide first exactly what it is you want the computer to do. Rough diagramming a "plan" of the program, referred to as a **flowchart**, helps get your thoughts together.

2. Start putting the instructions (referred to as source statements or code) down onto paper.

3. Enter the instructions into the computer through an **Editor program** which we will discuss in greater detail later.

4. Convert the source code you have written into object code that the computer can understand using an assembler program. If the assembler finds any errors, correct these and reassemble.

5. Run the program on the computer. If it performs differently than what you had expected, you must **debug** the program. This involves taking a copy of your source code and changing it until you can get the program to run right.

THE EDITOR PROGRAM

The Editor is a program that we have not mentioned yet. The Editor program allows you to write out your source code and edit it directly on the screen before assembling it. The Editor program also allows you to save an incomplete source program on disc for later revision. This book assumes that you are already familiar with the Editor program. If you are not sure, refer to the instructions in the beginning of the Editor/Assembler manual that accompanies the software. If you are using the mini-memory module and line-by-line assembler refer to chapter 10.

3.2 SOURCE CODE

Now that we have a general understanding about how to go about constructing source code, it is time to proceed along the specifics; namely creating a program.

The source code is a logical sequence of instructions designed to guide the computer along a desired course. A source statement can be categorized as an **instruction**, **pseudo-operation**, or an **assembler directive**.

As we have mentioned before, an assembly language abbreviation (instruction) is a symbolic representation of a binary instruction. It is translated literally by the assembler program during the assembly process.

Pseudo-operations and assembler directives give directions to the assembler program (not the computer) as to what to do with certain instructions or data.

Assembler directives, pseudo-operations and assembly language instructions will be covered in greater detail in future chapters.

CONSTANTS IN PROGRAMMING

When entering numbers or constants into the computer you may use one of several forms:

1. DECIMAL -- Entered as a base ten number. May be an unsigned number from 0 through 65535, or a signed value ranging from -32768 through 32767.

```
  123
-2410
65535
```

2. HEXADECIMAL -- Entered as a string of up to four alphanumeric (A thru F) characters preceded by a greater than (>) sign. The following are valid examples of hexadecimal constants:

```
>0F
>1AC
>32FD
```

3. CHARACTER CONSTANTS -- Entered as a string of ASCII characters enclosed in single quotes; for example 'A' or 'AD'. A character constant consisting of only two quotes (no characters) is also valid. The following are valid character constants:

Character Constant	ASCII values
'2'	'(50)'
'AB'	'(65)(66)'
'30%'	'(51)(48)(37)'
'HELLO !'	'(72)(69)(76)(76)(79)(32)(33)'

4. ASSEMBLY-TIME CONSTANTS -- These constants are defined at the time of assembly. The are written in the operand field of an EQU instruction. We will spend more time explaining how these constants are used when we reach this instruction in later chapters.

Negative numbers are also easily specified. If the constant is in decimal form simply precede it with a minus sign (e.g.-23). If the number is in hexadecimal notation you must enter it in its two's compliment form. For example, -42 and >D6 both represent the same value.

THE SOURCE STATEMENT

Each line in an assembly language program is referred to as a **source statement**. Each source statement contains up to four **Fields** separated by a single blank space. The fields are positioned as follows:

Label Op-code Operand(s) Comments

Of these four fields, only the **op-code** field is always required for a valid source statement. The other fields may or maynot be required depending on the op-code used. The maximum length of a source statement is 80 characters, however only 60 of these will be displayed when using a list file. The first character typed on a line begins the label field. If you do not use the label field then the first character must be a blank space. All the fields are separated by at least one blank space. The following is an example of a single source statement that uses all four fields:

```
MYREG   BSS  >32  *RESERVE MEMORY FOR MY WORKSPACE REGISTERS
```

The following sections will describe the four fields that make up a source statement.

LABEL FIELD

The **label field** is a name or label that you give to a source statement so that you can refer back to it. This label can then be used in other instructions to refer back to it. For example, when you instruct the computer to jump from one instruction to another, you give its destination by specifying its label.

Unless the first character is a blank, the first character in a source statement begins the label. It can be up to 6 characters in length. A label can be made up of any alphanumeric characters, but the first character must always be alphabetic. If you elect to omit the label field the first character of the source statement must be a blank space. Also, you are not allowed to put a blank space in the middle of a label; ie: MYREG not MY REG.

Labels are usually used to identify the target of a jump instruction.

OP-CODE FIELD

The **op-code field** (short for operation code) is also known as the **mnemonic field** (pronounced knee-mon-ik). It holds the one to four letter acronym for the microprocessor instruction. When the assembler program is run it uses an internal reference table to translate each acronym into the appropriate binary code. The type of op-code used determines how many and what type of operands should be found in the operand field.

OPERAND FIELD

The **operand field** contains the data or the location of the data needed by instruction in the op-code field. Some op-codes do not require an operand while others require one or more. If more then one operand is required they are separated by a comma. The operand field may contain one or more terms, expressions, or constants depending on the needs of the instruction in the op-code field.

To sum up, the operand field contains the data that the instruction in the op-code field refers to. For example, in this ADD operation:

```
A   R0,R1
```

the ADD (A) instruction refers to the addition of the value in Workspace Register 0 to the value in Workspace Register 1.

COMMENT FIELD

The **comment field** is an optional field that begins one space after the operand field ends. It is always begun with an asterisk (*). The comment fields contains comments written by the programmer as a reminder to what the source statement does. These statements are ignored by the assembler program during the assembly process.

Comments are utilized to remind you what the function of a source statement or group of source statements is. For example, the statement:

```
MYWSP  EQU  >8300  *BEGIN WORKSPACE AT THIS ADDRESS
```

reminds you that your workspace was begun at the specified address in memory. Comments can also stand alone on a line if the line begins with an asterisk (*). In this way entire blocks containing just comments can be constructed:

```
***********************************************
*                                             *
*              DEFINE EQUATES                 *
***********************************************
```

CHAPTER 3 STUDY EXERCISES

1. What is the name of the program that converts source code into object code?

2. A Software Workspace area consists of how many registers?

3. Which bit of the Status Register is set when the result of arithmetical expression is too large to be represented in two's compliment form?

4. Regarding the four fields in the source statement?

   (a) Which three of the fields may be optional?

   (b) Which character always begins a line of comment or is the first letter in a comment field.

   (c) Which field of a source statement is always required?

5. What is the difference between an assembler directive and a instruction.

6. Is the following a valid hexadecimal number?

   >DEFG

7. Which portion of a source statement determines which and what type of operands are required.

4

THE INSTRUCTION SET

For a quick review, remember that the location of each byte in RAM is designated by an address, much like the location of each house in a city is specified by its address. Also keep in mind that the number held at a particular address could in turn specify another address where information is located. With this in mind we will proceed with a discussion on **addressing modes** which simply stated, are ways of telling the assembler program exactly at what address in RAM needed information is located.

4.0 ADDRESSING MODES

Your Home Computer provides a variety of ways to access the numbers that your programs perform operations on. These numbers are referred to as **operands** and specific ways to address them are referred to as **addressing modes.** There are a total of five addressing modes available when programming, they are:

1. WORKSPACE REGISTER AND IMMEDIATE ADDRESSING
2. WORKSPACE REGISTER INDIRECT ADDRESSING
3. SYMBOLIC REGISTER ADDRESSING
4. INDEXED MEMORY ADDRESSING
5. WORKSPACE REGISTER INDIRECT AUTO-INCREMENT ADDRESSING

The operand is the actual value that is to be "operated on" by the instruction. How you want to specify the operand determines the addressing mode that you will use.

In the sections that follow each addressing mode is discussed in detail. An example is provided of each modes usage.

WORKSPACE REGISTER ADDRESSING

In **Workspace Register Addressing** The operand is located in the specified register. Remember that a Workspace consists of sixteen consecutive Registers labeled R0 through R15. Workspace Register 5 would thus be referred to as "R5". You specify in the beginning of your program where these registers will be located in RAM. We will have more on this later. An example of Workspace Register Addressing is the statement:

```
MOV   R2,R4
```

which moves a copy of the contents of Workspace Register 2 (R2) into Workspace Register 4 (R4). Another example:

```
A     R6,R7
```

adds the contents of Workspace Register 6 (R6) to the contents of Workspace Register 7 (R7). The result is then placed in R7.

When using Workspace Register Addressing Mode it is important to remember that the operand is found **in the Register specified.**

IMMEDIATE ADDRESSING

You can also specify a constant as a source operand. In this way the value is right there for the assembler to get and does not have to be located in a Register or found at another address. This is termed **Immediate Addressing.** An example is the following statement:

```
LI    R0,324
```

which places (loads) the value 324 into Workspace Register 0, and the statement:

```
LI    R9,>144
```

which loads the value >144 (324) into Workspace Register 9, and the statement:

```
LI    R6,-32
```

which loads the value -32 into Workspace Register 6.

NOTE: Remember when using signed numbers the most significant bit holds the sign of the number. This limits signed values to numbers that can be represented with only 15 bits. The signed values thus range from +32767 (>7FFF) to -32768 (>8000). Unsigned numbers, however can range from 0 (>0000) to 65535 (>FFFF) since bit 0 does not have to be used to hold the sign of the number.

INDIRECT ADDRESSING

With this type of addressing, the register specified contains the address of the operand instead of the operand itself. An indirect Workspace Register Address is preceded by an asterisk (*). For example, the statement:

```
MOV  *R3,*R0
```

copies the word at the address given in Workspace Register 3 into the address found in Workspace Register 0. Notice how both R3 and R0 are indirectly addressed, that is they both contain the address of the information rather then the information itself. Another example is the statement:

```
A  *R4,R6
```

which adds the contents of the word being held at the address given in Workspace Register 4 to the contents of the word in Workspace Register 6. The result is then placed in Workspace Register 6. Notice how in this case R4 is indirectly addressed while R6 is directly addressed.

INDIRECT AUTO-INCREMENT ADDRESSING

With this type of addressing the register specified contains the address of the operand as with indirect addressing. After the address is obtained from the Workspace Register, the address in the Workspace Register is incremented by 1 for a byte instruction or by 2 for a word instruction. This allows you to access data in memory in a sequential manner from a given starting point. A Workspace Register auto-increment address is preceded by an asterisk (*) and followed by a plus (+) sign. For example, the following statement:

```
A  *R3+,R1
```

adds the contents of the word found at the address given in R3 to the contents of R1. The result is placed in R1. The address in R3 is then incremented by two ('A' is a word instruction). Another example is the statement:

```
MOV R9,*R10+
```

which copies the contents of R9 into the address given in R10 and increments the address in R10 by two. Now lets consider an example using real values. Suppose R1 contains >0004 and R2 contains >000A and address >0004 contains the value >0010, then the statement:

```
A  *R1+,R2
```

adds the value found at address >0004 which is >0010, to the value found in R2 which is >000A. The result, >001A is placed in R2. The value in R1 is then incremented by two (A is a word instruction). Thus, after completion of this statement R1 contains >0006, and R2 contains >001A.

SYMBOLIC MEMORY ADDRESSING

This type of addressing allows you to use a symbol to represent the address that contains the operand. The symbolic memory address is preceded by an "at" character (@). For example, if R0 contains >0002 then the statement:

```
JOYI  EQU  >00FF
      .
      .
      .
      A    @JOYI,R0
```

adds the contents of R0 with the contents at "JOYI" (in this case >00FF) the result, >0101, would then be placed in R0. Another example is the statement:

```
MOV  @>AA03,@>0E3F
```

which copies the word at address >AA03 into location >0E3F.

INDEXED MEMORY ADDRESSING

With indexed addressing, the effective address is gotten by adding the value of an index register to a displacement variable. You often use this addressing mode to access elements in a table. In such a case the value in the index register points to the beginning of the table, and the displacement to an element in the table.

The indexed memory address is preceded by an "at" sign (@) after which comes the displacement value followed by the index register which is closed in parentheses. For example,

```
A    @4(R4),R1
```

gets the word found at the address computed by adding 4 to the address in R4. This word, in turn, is added to the word found in R1. The result is then placed R1. Another example in the statement:

```
MOV  R5,@TABLE+3(R7)
```

which copies the contents of register 5 into a memory word. The address of this memory word is determined by taking the sum of

TABLE plus 3 and adding it to the contents of register 7 (R7).

note: Workspace Register 0 (R0) is reserved and may not be specified as an index register.

PROGRAM COUNTER RELATIVE ADDRESSING

This addressing mode can only be used in the operand fields of "jump" instructions the program counter relative address is written as an expression that corresponds to an address at a word boundary. An Example is the statement:

```
JMP  GETKEY
```

which jumps unconditionally to location GETKEY. GETKEY is a label that you gave another source statement in the program.

It should be noted that when an expression (like GETKEY in the last example) is evaluated it is subtracted from the value of the current location plus two. This value is then divided by two with the result being placed in the object code. This value must fall between the values -128 through 127 or the jump will not be executed. This means that the destination of a jump cannot be any farther than 256 (>100) bytes from the current address in the program counter.

To sum up you are not allowed to make a jump (using JMP) in your program greater than >100 bytes in length.

ARITHMETIC OPERATIONS

When programming you will have occasion to add, multiply or otherwise manipulate numbers. The TMS9900 allows addition (+), subtraction (-), multiplication (*), and signed division (/).

When an expression is evaluated, the assembler first negates all constants or symbols preceded by a minus (-) sign. All succeeding operations are carried out from left to right. Precedence is only given to the negation of symbols and constants, not to any other procedure. Therefore 4+6/2 is evaluated as 5 and not as 7. A remainder is disregarded in division, thus 5/2+4 equals 6.

Parentheses cannot be used to alter the order that an expression is evaluated in.

4.2 THE INSTRUCTION SET

The TMS9900 recognizes a number of different instructions. Table 4.1 lists the assembler mnemonic for each instruction and explains what each mnemonic stands for. Also listed is the required operand(s) and operand format for each instruction. You should

have a thorough understanding of addressing modes before proceeding to the instruction set.

TABLE 4.1 INSTRUCTION SET

MNEMONIC	DESCRIPTION	OPERAND(S) & FORMAT
A	ADD WORDS	G,(G)
AB	ADD BYTES	G,(G)
ABS	TAKES ABSOLUTE VALUE OF OPERAND	G
AI	ADDS AN IMMEDIATE VALUE TO WORKSPACE REG.	(W),#
ANDI	LOGICAL AND IMMEDIATE VALUE	(W),#
B	BRANCH	G
BL	BRANCH & LINK	G
BLWP	BRANCH & LINK WORKSPACE POINTER	G
C	COMPARE WORDS	G,G
CB	COMPARE BITS	G,G
CI	COMPARE IMMEDIATE VALUE	W,#
CLR	CLEAR	G
COC	COMPARE ONES CORRESPONDING	G,W
CZC	COMPARE ZEROS CORRESPONDING	G,W
DEC	DECREMENT	G
DECT	DECREMENT BY TWO	G
DIV	DIVIDE	G,W
INC	INCREMENT	G
INCT	INCREMENT BY TWO	G
INV	INVERT	G
LDCR	LOAD CRU	G,#*
LI	LOAD IMMEDIATE VALUE	(W),#
LIMI	LOAD INTERRUPT MASK WITH IMMEDIATE VALUE	#
LWPI	LOAD WORKSPACE POINTER W/ IMMEDIATE VALUE	#
MOV	MOVE	G,(G)
MOVB	MOVE BYTE	G,(G)
MPY	MULTIPLY	G,(W)
NEG	NEGATE	G
ORI	LOGICAL OR IMMEDIATE VALUE	(W),#
RTWP	RETURN WORKSPACE POINTER	
S	SUBTRACT	G,(G)
SB	SUBTRACT BYTES	G,(G)
SBO	SET CRU BIT TO ONE	CRU
SBZ	SET CRU BIT TO ZERO	CRU
SETO	SET TO ONE	G
SLA	SHIFT LEFT ARITHMETIC	(W),#**
SOC	SET ONES CORRESPONDING	G,(G)
SOCB	SET ONES CORRESPONDING, BYTE	G,(G)
SRA	SHIFT RIGHT ARITHMETIC	(W),#**
SRC	SHIFT RIGHT CIRCULAR	(W),#**
SRL	SHIFT RIGHT LOGICAL	(W),#**

TABLE 4.1 INSTRUCTION SET (CONTINUED)

MNEMONIC	DESCRIPTION	OPERAND(S) & FORMAT
JEQ	JUMP IF EQUAL	P
JGT	JUMP IF GREATER THAN	P
JH	JUMP IF LOGICAL HIGH	P
JHE	JUMP IF HIGH OR EQUAL	P
JL	JUMP IF LOGICAL LOW	P
JLE	JUMP IF LOW OR EQUAL	P
JLT	JUMP IF LESS THAN	P
JMP	JUMP	P
JNC	JUMP IF NO CARRY	P
JNE	JUMP IF NOT EQUAL	P
JNO	JUMP IF NO OVERFLOW	P
JOC	JUMP ON CARRY	P
JOP	JUMP IF ODD PARITY	P
STRC	STORE CRU	(G),#*
STST	STORE STATUS	W
STWP	STORE WORKSPACE POINTER	W
SWPB	SWAP BYTES	G
SZC	SET ZEROS CORRESPONDING	G,(G)
SZCB	SET ZEROS CORRESPONDING, BYTE	G,(G)
TB	TEST CRU BIT	CRU
X	EXECUTE	G
XOP	EXTENDED OPERATION	G,#***
XOR	EXCLUSIVE OR	G,(W)

* This operand represents the number of bits to be transferred. This value ranges from 0 through 15 with 0 indicating 16 bits.
** This operand is the shift count.
***This operand specifies the extended operation.

G - Indicates a general address which can be in one of any of the following modes:
- a) Workspace Register
- b) Indirect Workspace Register
- c) Symbolic Memory
- d) Indexed Memory Address
- e) Indirect Workspace Register Auto-Increment

W - When this is specified the operand has to be a Workspace Register Address.
\# - Value entered as a constant.
P - This operand is a program counter relative address.
CRU - Give CRU bit address.
( ) - The address at which a result is placed when two operands are required.

The instruction set can be divided into the following 7 functional groups:

1. DATA TRANSFER INSTRUCTIONS or LOAD AND MOVE Instructions allow you to move information between registers and memory locations.
2. ARITHMETIC INSTRUCTIONS allows you to perform arithmetic operations.
3. JUMP AND BRANCH INSTRUCTIONS performs jumps, calls to procedures, and returns from procedures, in this way you can control the order in which the program executes.
4. COMPARE INSTRUCTIONS let you compare words, bytes, or immediate values with each other. They even let you compare groups of bits within a byte or word that may correspond with each other. These instructions only affect the Status Register.
5. LOGICAL INSTRUCTIONS permit the performance of logical operations on workspace registers and memory locations.
6. SHIFT INSTRUCTIONS allow you to shift the bits in a Workspace Register a specified number of positions.
7. BIT INPUT/OUTPUT INSTRUCTIONS allow you to manipulate the CRU bits.

4.3 DATA TRANSFER INSTRUCTIONS

Data transfer instructions move numbers between registers and memory locations. Table 4.2 outlines the format of each instruction as well as which bits of the Status Register are affected by each instruction.

TABLE 4.2

		Status Register Bits (x) indicates bits affected by instruction										
MNEMONIC	FORMAT	L>	A>	EQ	C	OV	OP	X	INT		MASK	
MOV	G,(G)	X	X	X	-	-	-	-	-	-	-	-
MOVB	G,(G)	X	X	X	-	-	X	-	-	-	-	-
LI	(W),#	X	X	X	-	-	-	-	-	-	-	-
LWPI	#	-	-	-	-	-	-	-	-	-	-	-
LIMI	#	-	-	-	-	-	-	-	X	X	X	X
STST	W	-	-	-	-	-	-	-	-	-	-	-
STWP	W	-	-	-	-	-	-	-	-	-	-	-
SWPB	G	-	-	-	-	-	-	-	-	-	-	-

(MOV) MOVE WORD

One of the foundational instructions in assembly language is the "move word" (MOV) instruction. It can transfer a word from a source operand into a destination operand. The destination operand is then compared to zero and sets (or resets) the L>, A>, and EQ status bits accordingly.

The following are examples of operand combinations that are legal:

```
MOV  @HERE,@THREE   *MEMORY TO MEMORY (COPY INTO THERE)
MOV  @HERE,R7       *MEMORY TO REGISTER (LOAD REGISTER)
MOV  R3,R4          *REGISTER TO REGISTER
MOV  R7,@DEST       *REGISTER TO MEMORY
```

Another use of the MOV instruction is to compare a memory location to zero. For example, the following source statements:

```
MOV  R5,R5          *Move R5 into itself and compares it to 0.
JEQ CHECK           *Jump to location "CHECK" if R5=0.
```

move Workspace Register 5 into itself and then compares the contents of R5 to zero. If R5 is equal to zero than the EQ bit is set and the JEQ instruction will cause the program to "jump" to location "CHECK".

(MOVB) MOVE BYTE

This instruction copies the most significant byte of the source operand into the destination operand. For example suppose memory location >2E32 contains the value >23A6 and HOLD is located at address >2E32, and if R2 contains >34CC then the statement:

```
MOVB @HOLD,R2
```

changes the contents of R2 to >23CC and compares the contents of R2 to zero. As a result of this comparison and the logical greater than, arithmetic greater than, and odd parity bits are set, while the equal status bit is reset.

(LI) LOAD IMMEDIATE

Places a given number in a specified Workspace Register. The contents of this register is compared with zero and the results of this comparison affect the L>, A>, EQ bits of the Status Register accordingly. For example, the statement:

```
LI  R2,>23    *Load Workspace Register 2 with >0023.
```

loads R2 with >0023 (35) and sets the logical greater than, arithmetic greater than, and resets the equal status bits.

(LWPI) WORKSPACE POINTER IMMEDIATE

Places the Workspace Pointer at the address specified by the immediate operand. For example, the statement:

```
START LWPI >20BA  *SET START EQUAL TO >20BA
```

Sets START equal to >20BA and also sets the Workspace Pointer to location >20BA. The LWPI instruction has no effect on the Status Register.

(LIMI) LOAD INTERRUPT MASK IMMEDIATE

This instruction loads the interrupt mask of the Status Register (bits 12-15) with a specified value. For example, the statement:

```
LIMI  2
```

sets the interrupt mask at 2 (>0010) and enables interrupts at levels 0, 1, and 2. While the statement:

```
LIMI 0
```

disables all interrupts and is the normal state of the computer (>0000).

(STST) STORE STATUS REGISTER

Stores the current contents of the Status Register in a specified Workspace Register. For example the statement:

```
STST R5
```

stores the current Status Register contents in Workspace Register 5.

(STWP) STORE WORKSPACE POINTER

This instruction saves a copy of the contents of the Workspace Po.nter Register in a specified Workspace Register. For example, thc statement:

```
STWP  R4
```

stores the Workspace Pointer value in R4.

(SWPB) SWAP BYTES

This instruction switches the most significant byte with the least significant byte in a General Register. In other words, SWPB

exchanges the left and right bytes of a specified word. For example, the statement:

```
SWAP  SWPB  R2
```

replaces the most significant byte of register 2 (bits 0-7) with a copy of the least significant byte (bits 8-15) contained within the register. Conversely, the least significant byte of register 2 is replaced with a copy of the most significant byte. In this way bytes can be interchanged in anticipation of various byte instructions. In another example, suppose R0 contained the value >2244, and memory location >2244 contained the value >FF33, the instruction:

```
SWPB  *R0
```

would change the contents of memory location >2244 to >33FF.

In summary, the SWPB instruction exchanges left and right (least/most significant bytes) of a word specified in a general register. The SWPB instruction has no effect on the Status Register.

4.4 THE ARITHMETIC INSTRUCTIONS

Arithmetic instructions allow you to perform a variety of arithmetic operations in your program. Table 4.3 shows which bits of the Status Register that are affected by each instruction.

TABLE 4.3 ARITHMETIC INSTRUCTIONS

		STATUS REGISTER BITS (X) INDICATES BITS AFFECTED BY INSTRUCTION										
Mnemonic	Format	L>	A>	EQ	C	OV	OP	X	INT	MASK		
A	G,(G)	X	X	X	X	X	-	-	-	-	-	-
AB	G,(G)	X	X	X	X	X	X	-	-	-	-	-
ABS	G	X	X	X	-	X	-	-	-	-	-	-
AI	(W),#	X	X	X	X	X	-	-	-	-	-	-
DEC	G	X	X	X	X	X	-	-	-	-	-	-
DECT	G	X	X	X	X	X	-	-	-	-	-	-
DIV	G,(W)	-	-	-	-	X	-	-	-	-	-	-
INC	G	X	X	X	X	X	-	-	-	-	-	-
INCT	G	X	X	X	X	X	-	-	-	-	-	-
MPY	G,(W)	-	-	-	-	-	-	-	-	-	-	-
NEG	G	X	X	X	-	X	-	-	-	-	-	-
S	G,(G)	X	X	X	X	X	-	-	-	-	-	-
SB	G,(G)	X	X	X	X	X	X	-	-	-	-	-

p. 37 ↔ after p 48

(A) ADD WORDS

This instruction adds a copy of the source operand to a copy of the destination operand and places the sum in the destination operand. For example, the statement:

```
A  *R3,*R4+
```

adds the contents of the word found at the address in R3 to the word found at the address in R4. The sum is placed at the address given in R4 and the address in R4 is incremented by two (Workspace Register auto-increment addressing). The sum is compared to zero and the results of the comparison are reflected in the Status Register. Another example we can look at supposes that the address labeled TABLE contains >2123 and R2 contains >000B, the statement:

```
A  R2,@TABLE
```

then causes the contents at TABLE to change to >212E. The logical and arithmetic greater than bits are set and the equal, carry and overflow bits are reset in the Status Register. The contents of R2 remain >000B.

(AB) ADD BYTES

This instruction adds the left most byte (bits 0-7) of the specified source register to the left most byte of the destination register. The result is placed in the left-most byte of the destination register. For example, in the statement:

```
AB  R3,R4
```

the left byte of R3 is added to the left byte of R4 and the sum is placed in the left byte of R4. Another example, suppose that R2 contained the address >23FA at which was located the memory word >2233, and R3 contains >DD88, then the statement:

```
AB  *R2+,R3
```

changes the contents of R3 to >FF88 and increments R2 by one to >23FB. This result is obtained by taking the left most byte of the memory word specified in the address given in R2 (>22) and summing it with the left most byte in R3 (>DD) coming up with >FF. This sum is then placed in the left most byte of R3 and R2 is incremented to >23FB. Comparison of the sum with zero sets the logical greater then, overflow, and odd parity bits of the Status Register while it resets the arithmetic greater than, equal, and carry status bits. Another example, R4 contains >8100, and

The instruction is ignored and the overflow status bit is set while the source and destination operands remain unchanged. Lets take some time now to look at a few examples to see if we can clarify things. Suppose that memory location LOCA contains >0005, R2 contains >0001 and R3 contains >000D, then the statement:

```
DIV @LOCA,R2
```

divides 65549 (>0001000D) by 5 and places the quotient 13109 (>3335) in R2 and the remainder, .2 (represented as "2") in R3. In another example suppose that LOCA contains >0002 and R2 contains >0004, also R3 contains a zero, then the statement:

```
DIV @LOCA,R2
```

attempts to divide 262144 (>00040000) by 2. The resultant quotient, 131072, cannot be represented in a 16-bit word. The result is that the overflow bit is set in the Status Register and the operation is canceled.

In summary, the destination operand is a consecutive 2-word area of a Workspace Register. It should be noted that if the destination operand is Workspace Register 15 (R15) the first word of the destination operand is in R15 and the second word is in the memory location immediately following the Workspace area.

Note that the DIV instruction does not let you divide by an immediate value directly. To do this, you must put the immediate value into a register a Register or memory location. The following examples illustrates this point.

```
HERE  EQU  >14          *
THERE EQU  >05          * Load equates
ZERO  EQU  >00          *
      .
      MOV  @HERE,R7     * Move
      MOV  @THERE,R5    *     values  into
      MOV  @ZERO,R6     *                  Registers
      DIV  R5,R6        * Computes 20/5, result goes in R6.
```

Another example,

```
      LI   R5,>05       *
      LI   R6,0         * Load Registers
      LI   R7,>14       *
      DIV  R5,R6        * Computes 20/5, result goes in R6.
```

(INC) INCREMENT

This instruction increments the source operand by one (1). The result then replaces the source operand. The computer compares the new value to zero and sets/resets the status bits accordingly.

With a carry of bit 0, the carry bit is set. With an overflow, the overflow bit is set. An example of the INC instruction is the statement:

```
INC  @ADRS
```

which increments the value specified at location ADRS by one.

(INCT) INCREMENT BY TWO

This instruction increments the source operand by two (2). The result then replaces the source operand. The computer then compares the sum to zero and sets/resets the status bits accordingly. When there is a carry of bit zero the carry bit is set. With an overflow, the overflow bit is set. Lets consider an example where R3 contains >0022:

```
INCT  R3
```

this statement then increments R3 by two and places the result (>0024) in R3. The arithmetic greater than, logical greater than status bits are set while the equal, carry, and overflow status bits, are reset.

Both the increment and the decrement instructions are useful to index byte arrays while the increment and decrement by two instructions are useful to index word arrays.

(MPY) MULTIPLY

The MPY instruction performs a multiplication. The source operand is multiplied by the destination operand. The product is then placed in the 2-word destination operand. For example if R0 contains the value >0003, R3 contains the value >0005, and R4 contains the value >0EA7, the statement:

```
MPY  R0,R3
```

multiplies the contents of R0 and R3 together to get >000F and places this value in R4. R3 now contains a zero (>0000). The Status Register is unaffected by the MPY instruction. Another example supposes that the memory location HERE contains >FFFF and R3 contains >0002, then the statement:

```
MPY  @HERE,R3
```

multiplies the contents HERE (65535) to the contents of R3 (2). The product 131070 (>0001FFFE) is placed into R3 (>0001) and R4 (>FFFE). Memory location HERE is unchanged as is the Status Register. If the destination operand is specified as R15 the product is placed into R15 and the first memory word immediately following the workspace memory area.

(NEG) NEGATE

This instruction replaces the source operand with its additive inverse. The computer then compares the result to zero and sets/resets the status bits to reflect this comparison. Suppose memory location VALUE1 contains the value >9BC1, then the statement:

```
NEG @VALUE1
```

changes the contents of VALUE1 to >643E. The logical greater than and arithmetic greater than status bits are set in the Status Register while the equal and overflow status bits are reset.

(S) SUBTRACT WORDS

This instruction subtracts a copy of the source operand from a copy of the destination operand and places the result in the destination operand. The result is compared to zero and the status bits are set/reset accordingly. When there is a carry of bit zero, the carry bit is set. When there is an overflow, the overflow bit is set. For example, suppose that memory location HERE contains >2123 and memory location THERE contains >AA33, then the statement:

```
S  @HERE,@THERE
```

changes the contents of THERE to >8E10 (>AF33->2123). The logical greater than, arithmetic greater than, carry and overflow status bits are set, while the equal status bit is reset.

(SB) SUBTRACT BYTES

This instruction subtracts the source operand, which is a single byte, from the destination operand, which is also a single byte. The difference is then placed in the destination operand. The computer compares the resulting byte to zero and sets/resets the Status Register bits to reflect the results of this comparison. When there is a carry of the most significant bit of the byte (bit 0), the carry status bit is set. When there is an overflow, the overflow status bit is set. If the resulting byte has an odd number of bits set to one, then the odd parity bit is set. If the operand is specified in a Workspace Register, then only the left most bits (bits 0-7) are used. For example the statement:

```
SB  R0,R1
```

which subtracts the left-most byte of R0 from the left-most byte of R1, and places the difference in the leftmost byte of R1. Another example supposes that memory location ADDR contains the value >131D and R5 contains the value >23F5, then the statement:

```
Sb  R5,@ADDR
```

changes the contents of R5 to >F610. The logical greater than bit is set, while the other status bits affected by this instruction are reset.

4.5 JUMP & BRANCH INSTRUCTIONS

Jump instructions as well as branch instructions are used to transfer control from one area of the program to another. This control transfer may be conditional or nonconditional. These instructions are mainly used to control the sequence in which a program executes. Table 4.4 outlines the conditional and nonconditional branch and jump instructions and the status bits tested by each instruction:

TABLE 4.4 JUMP & BRANCH INSTRUCTIONS

Mnemonic	Format	STATUS REGISTER BITS TESTED/AFFECTED (t) indicates bits tested by instruction L>	A>	EQ	C	OV	OP	X	INT MASK
		UNCONDITIONAL TRANSFERS							
B	G	-	-	-	-	-	-	-	- - - -
BL	G	-	-	-	-	-	-	-	- - - -
BLWP	G	-	-	-	-	-	-	-	- - - -
JMP	expression	-	-	-	-	-	-	-	- - - -
RTWP		*x	x	x	x	x	x	x	x x x x
		CONDITIONAL TRANSFERS							
JEQ	expression	-	-	t	-	-	-	-	- - - -
JNE	expression	-	-	t	-	-	-	-	- - - -
JH	expression	t	-	t	-	-	-	-	- - - -
JL	expression	t	-	t	-	-	-	-	- - - -
JHE	expression	t	-	t	-	-	-	-	- - - -
JLE	expression	t	-	t	-	-	-	-	- - - -
JGT**	expression	-	t	-	-	-	-	-	- - - -
JLT**	expression	-	t	t	-	-	-	-	- - - -
JNC	expression	-	-	-	t	-	-	-	- - - -
JOC	expression	-	-	-	t	-	-	-	- - - -
JNO	expression	-	-	-	-	t	-	-	- - - -
JOP	expression	-	-	-	-	-	t	-	- - - -
		ITERATION CONTROLS							
X	source	***x	x	x	x	x	x	x	x x x x
XOP	source,operation	-	-	-	-	-	-	-	- - - -

t=tested status bit, x=affected status bit

* Restores all status bits to the value contained in Workspace Register 15 (R15).

** Only JGT & JLT instructions use signed arithmetic comparisons. All other comparisons are logical (unsigned) comparisons.

*** The instruction 'X' does not directly affect any status bits, however the executed instruction affects the Status Register accordingly.

(B) BRANCH

This instruction transfers control to another line in the program. It does this by replacing the contents of the Program Counter Register with the address specified in the operand. This instruction has no affect on the Status Register. For example, if R4 contains >32F1, the statement:

```
B  *R4
```

causes the word at location >32F1 to be placed in the Program Counter Register. This has the effect of letting the word at location >32F1 be used as the next instruction executed by the program.

(BL) BRANCH AND LINK

This instruction transfers control to another line in the program. It also stores the address of the instruction immediately following the BL in R11. The transfer of control is accomplished by replacing the value in the Program Counter Register with the value specified by the source operand. The BL instruction has no affect on the Status Register. For example, if the statement:

```
BL  @SUBL
```

occurs at memory location >06CA, the instruction places the value >06CE in R11 and places memory location SUBL in the Program Counter Register.

note: The instruction BL @SUBL requires two words of machine code which are placed at addresses >06CA and >06CC. Therefore, the word address immediately following the second word is >06CE which is the value placed in R11.

(BLWP) BRANCH AND LOAD WORKSPACE POINTER

When this instruction is implememnted the following occurs:

1) The source operand is placed in the Workspace Pointer Register.

2) The word immediately following the source operand is placed in the Program Counter Register.

3) The previous contents of the Workspace Pointer Register are placed in new Workspace Register 13 (R13).

4) The previous contents of the Program Counter Register (the address of the instruction immediately following BLWP) are placed in new Workspace Register 14 (R14).

5) The contents of the Status Register are placed in the new Workspace Register 15 (R15).

When all operations are finished, the computer transfers control to the new value in the program counter. With the BLWP instruction you can link to subroutines and program modules that do not necessarily share the calling programs workspace.

(JMP) UNCONDITIONAL JUMP

The JMP instruction allows you to move around in your program. It is similar to the GOTO instruction in BASIC. The JMP instruction causes the computer to take its next instruction from another location. It does not affect the Status Register. It's clean and simple. The following are examples of the JMP instructions usage:

```
1)  JMP  THERE  * Jumps to location THERE.
2)  JMP  >11AF  * Jumps to address >11AF.
```

Keep in mind that when using 'jump' instructions the address you are jumping to has to be within >100 bytes of the address of the jump instruction or the instruction is ignored.

(RTWP) RETURN WORKSPACE POINTER

This instruction serves to return the computer to how things were before the calling of a subroutine through use of a BLWP instruction. Also returns from an interrupt or XOP instruction. The RTWP instruction accomplishes this in the following steps:

1) Replaces the contents of the Workspace Pointer with a copy of R13.

2) Replaces the value in the Program Counter Register with a copy of R14.

3) Replaces the contents of the Status Register with a copy of R15.

In summary, the RTWP instruction restores the execution environment after completion of a BLWP instruction, interrupt, or XOP instruction.

CONDITIONAL TRANSFERS

There are 12 different instructions that allow your computer to make a "decision" before proceeding along a course of action. These decisions are based on the contents of the Status Register. Some of the conditional jump instructions test to see if the carry (C) bit has been set others test differing combinations of bits. For instance, the instruction Jump on Odd Parity (JOP) jumps only when the Odd Parity (OP) bit is set, others such as the Jump if logical High (JH) only jump if the logical greater than (L>) bit is set to 1 and the equal (EQ) bit is reset to 0.

The conditional jump instructions do not change any of the status bits; instead they are the instructions which look at the bits in the Status Register. They are the only instructions which base their activity on Status Register settings. **They are the reason the Status Register exists at all.**

All conditional transfer "jump" instructions occupy 2 bytes in memory. The first byte holds the operation code, while the second holds the relative displacement. You should always try and construct your programs so that the expected outcome executes when the jump is not taken.

Here are a few examples of conditional transfer "jump" instructions:

```
1)  A     R0,R1   * Jumps to location BIG if the add instruction
    JOC   BIG     * produces a carry.

2)  S     R4,R5   * Jumps to location ZERO if the result of this
    JEQ   ZERO    * subtraction operation is a 0.
```

You can also check to see if a Register contains a zero by using a MOV instruction, as in the following example:

```
3)  MOV   R4,R4   * Copies the contents of R4 into itself and
    *               compares the result to zero.
    JEQ   ZERO    * Jump to location ZERO if EQ bit set.
```

You can also set up a counter in a program for use in creating delays, loops, arrays, or printing to consecutive screen locations. Counters have the general format:

```
4)        LI   R1,1000   * Put 1000 in R1.
   DELAY  DEC  R1        * Decrement R1.
          MOV  R1,R2     * Copy R1 into R1 and compare R1 to 0.
          JNE  DELAY     * Jump if R1>0 (EQ=0) to DELAY.
          .              * Continue program.
```

Conditional jump instructions have the general format:

J-- expression

where (--) is a one or two letter modifier. The expression may be a constant or symbol. Looking at Table 4.5 we see a summary of the conditioina^ jump instructions, as well as the conditions that cause a jump to occur. 'jump...if' refers to status bit settings.

TABLE 4.5 CONDITIONAL JUMP INSTRUCTIONS

Instruction	Description	'Jump if....'
JEQ	JUMP IF EQUAL TO ZERO	EQ=1
JNE	JUMP IF NOT EQUAL TO ZERO	EQ=0
JH	JUMP IF LOGICALLY HIGHER THAN ZERO	L>=1 & EQ=0
JL	JUMP IF LOGICALLY LOWER THAN ZERO	L>=0 & EQ=0
JHE	JUMP IF LOGICALLY HIGH OR EQUAL TO ZERO	L>=1 or EQ=1
JLE	JUMP IF LOGICALLY LOW OR EQUAL TO ZERO	L>=0 or EQ=1
JGT*	JUMP IF GREATER THAN ZERO	A>=1
JLT*	JUMP IF LESS THAN ZERO	A>=1 & EQ=0
JNC	JUMP ON NO CARRY (CARRY BIT RESET)	C=0
JOC	JUMP ON CARRY (CARRY BIT SET)	C=1
JNO	JUMP IF NO OVERFLOW	OV=0
JOP	JUMP IF ODD PARITY	OP=1

*signed comparisons/all others use unsigned (logical) comparisons

(X) EXECUTE

The execute instruction allows you to utilize a source operand as an instruction. The X instruction does not alter the Status Register, but the inserted instruction affects status bits normally. If a jump is executed (that is if the status test for a jump is passed) the jump is executed from the location of the X instruction. The X instruction can specify an instruction one, two, or three words in length. Th^ Program Counter Register is then incremented the required one, two, or three words required by the source operand. The X instruction is mainly used when the instruction needed is dependent upon a variable factor.

4.6 COMPARISONS

It is very useful to compare various values when computing. That is the purpose of the compare instruction set. Compare instructions have no effect other then to set or reset various status bits. They are used in combination with conditional jump instructions to help the program make decisions. The compare instructions make simultaneous logical and arithmetic comparisons.

Arithmetic comparisons compare the two operands as two's compliment values. A logical comparison compares them as unsigned numbers. Table 4.6 outlines a summary of the compare instructions and the status bits each affect:

TABLE 4.6 COMPARE INSTRUCTION SET

Mnemonic	Format	(x) indicates bits affected by instruction: Status Register Bits L>	A>	EQ	C	OV	OP	X	INT	MASK		
C	G,G	x	x	x	-	-	-	-	-	-	-	-
CB	G,G	x	x	x	-	-	x	-	-	-	-	-
CI	W,#	x	x	x	-	-	-	-	-	-	-	-
COC	G,W	-	-	x	-	-	-	-	-	-	-	-
CZC	G,W	-	-	x	-	-	-	-	-	-	-	-

(C) COMPARE WORDS

This instruction compares the source operand, which is a word of memory, with the destination operand which is also a word of memory. The result of this comparison is then reflected by the Status Register. The arithmetic greater than and equal status bits reflect a signed comparison while the logical status bit reflects an unsigned (16 bit) comparison. The operands are left unchanged.

The compare instructions act very much like the subtract (S) instruction in that the compare instructions subtract a source operand from a destination operand. The difference is then compared with zero with the Status flags being set accordingly. But unlike the Subtraction instruction, the compare instructions do not save the result; the operands remain unchanged. The sole function of the compare instructions is to set/reset status bits in the Status Register for decision-making by conditional jump instructions. An example of the compare word (C) instruction is the statement:

```
C  R0,R1
```

which compares R0 with R1 (the contents of R0 are subtracted from the contents of R1; the difference being compared to zero). The Status Register is then set/reset to reflect the result of the comparison. Table 4.7 gives some examples of the compare words (C) instruction:

TABLE 4.7 COMPARE WORDS INSTRUCTION

		Status Register settings after 'C' instruction		
Source-op	Destination-op	Logical (L>)	Arithmetic (A>)	Equal
>FFFF	>0000	1	0	0
>7FFF	>0000	1	1	0
>8000	>0000	1	0	0
>3FB2	>3FB2	0	0	1
>F214	>B345	1	0	0
>6000	>5FFF	0	0	0

(CB) COMPARE BYTES

The compare bytes instruction is very similar to the compare words instruction we have just covered. The exception is that two bytes are compared instead of two words. For example, the statement:

```
CB   R0,R1
```

compares the left byte of R0 with the left byte of R1. The result of this comparison will set or reset the appropriate status bits. The operands are unaffected. In addition to the L>, A>, and EQ status bits, the OP (Odd Parity) status bit is set when the result of the CB operation (really a subtraction operation) contains an odd number of logic one bits. Table 4.8 gives some examples of the use of the CB instruction:

TABLE 4.8 COMPARE BYTES INSTRUCTION

		Status Register settings after 'CB' instruction			
Source-op	Destination-op	L>	A>	Equal	Odd Parity
>00	>FF	1	0	0	0
>00	>7F	1	1	0	1
>7F	>80	1	0	0	1
>7F	>7F	0	0	1	0
>80	>7F	0	1	0	1

address >2232 contains >F411. Also R5 contains >2233, the statement:

```
AB  R4,*R5
```

then changes the memory word >2232 to >F492 because >81 (the value of the left most byte in R4) plus >11 the (the value in memory byte >2233) equals >92. The left byte in memory word >2232 is unchanged. In this example the logical greater than, overflow, carry and odd parity bits are set, while the arithmetic greater than, and equal bits are reset.

(ABS) ABSOLUTE VALUE

This instruction takes the absolute value of an operand. It first checks the sign bit (bit 0) to see if it is equal to one. It it is then the two's compliment of the number is taken. If the sign bit is equal to zero, then the number is already positive and the source operand is unchanged. For instance, if R0 contains the value >FE00 then the statement:

```
ABS  R0
```

changes the value of R0 to >0020. In this case when the result is compared to zero, the logical greater than and arithmetic greater than status bits are set, while the overflow, and equal status bits are reset.

(AI) ADD IMMEDIATE

This instruction adds an immediate value to a specified Workspace Register and places the result in the Workspace Register. The sum is then compared with zero and the Status Register bits are set/reset accordingly. For example, the statement:

```
AI  R2,8
```

adds the value 8 to the contents of R2 and places the result in R2. Another example supposes that R5 contains >0006:

```
AI  R5,>23
```

the value >0029 is placed in R5. In this case the logical greater than and arithmetic greater than status bits are set, while the equal, carry, and overflow bits are reset.

(DEC) DECREMENT

This instruction decrements the contents of a specified general register (or a memory location specified in the address) by one (1). The result then replaces the source operand. The result is

compared with zero setting/resetting the Status Register accordingly. For example, the statement:

```
DEC  *R4
```

decrements by one, the word starting at the address given in R4. The DEC instruction is very helpful in counting and indexing of **byte arrays.** For example, if memory location TABLE contains the value >0001, then the statement:

```
DEC @TABLE
```

places a value of zero in location TABLE (>0000). As a result of this the equal and carry status bits are set, while the logical greater than, and overflow status bits are reset.

(DECT) DECREMENT BY TWO

This instruction decrements the source operand by two (2). The result then replaces the operand. For example, the statement:

```
DECT  @ADDR1
```

decrements the contents of ADDR1 by two. The result is compared to zero with the results of this comparison setting or resetting the status bits accordingly. The carry bit is set if there is a carry of bit zero. The DECT instruction is very helpful in counting and indexing **word arrays.** For instance, suppose memory location TABLE contains the value >2AE0 then the statement:

```
DECT  @TABLE
```

places a value of >2ADE in TABLE. The logical greater than, arithmetic greater than and carry status bits are set, while the equal and overflow status bits are reset.

(DIV) DIVIDE

This instruction divides the destination operand (which is a consecutive two (2) word area of a Workspace Register) by a copy of the source operand (one word from a general Register). For example the instruction:

```
DIV  R1,R2
```

divides the contents of Workspace Registers 2 and 3 by the contents in Workspace Register 1. It should be remembered that when the source operand is greater than the destination operand, normal division occurs. However, if the source operand is less than or equal to the first word in the destination operand, then the quotient will be too large to be represented in a 16 bit word.

(CI) COMPARE IMMEDIATE

This instruction compares the contents of a Workspace Register to some immediate value. For example, the statement:

```
CI  R3,>21
```

and the statement:

```
CI  R3,33
```

both compare the contents of R3 with the number 33. The comparison is accomplished in the same manner as with the C instruction. The Status Register bits are set/reset to reflect the results of the comparison.

(COC) COMPARE ONES CORRESPONDING

This instruction will set the EQ status bit if the bit positions set to 1 in the destination operand correspond to the bit positions set to one in the source operand. For example, the statement:

```
COC  @TEST,R3
```

compares the logic bits set to 1 in TEST with the bits set to 1 in R3. In another example, suppose MASK contains the word >D012 and R3 contains the value >F893, then the statement:

```
COC  @MASK,R3
```

sets the equal status bit to 1 for we see that:

```
>D012  =  1101  0000  0001  0010 and
>F893  =  1111  1000  1001  0011
```

for each bit set to 1 in the source operand there is a 1 bit in the corresponding bit position of the second operand. If R3 had contained >F890, the equal status bit would have been reset.

(CZC) COMPARE ZEROS CORRESPONDING

This instruction will set the EQ status bit if the bits in the source operand that are set to 1 correspond to the bits set to 0 in the destination operand. For example, the statement:

```
COC  @MASK,R3
```

compares the bits set to 1 in MASK, with the bits set to zero (0) in R3. In another example, suppose MASK contains the word >AB32, and R3 contains the value >44DF, the above instruction sets the

equal (EQ) status bit to 1 because:

```
>AB32  =  1010  1011  0011  0010 and
>44DF  =  0100  0100  1101  1111
```

for every logic bit set to 1 (one) in the source operand (>AB32), there is a logic bit set to (0) zero in the corresponding bit position of the destination operand (>44DF). However, if R3 had contained the value >44DE the EQ bit would have been reset because:

```
>AB32  =  1010  1011  0011  0010 and
>44DE  =  0100  0100  1101  1110
```

in the destination operand (>44DE), in bit position 15 (least significant bit) the bit is not set (not=1).

The COC and CZC instructions are used to compare a word with a mask in order to see if either its one bits correspond or its zero bits correspond. To sum up, the COC instruction is used to determine if the word in a Workspace Register has 1's that correspond to the 1's in a mask that you specify. Conversely, the CZC instruction is used to determine of the word in a Workspace Register has 0's in the bit positions indicated by 1's in a specified mask.

4.7 LOGICAL INSTRUCTIONS

Logical instructions are so named because they operate according to the rules of formal logic as opposed to the rules of mathematics. When dealing with logical instructions it is helpful to think in terms of bits set (=1) as "true" and bits reset (=0) as "false." There are ten instructions that allow you to perform various logical operations on memory locations and/or Workspace Registers. These instructions are outlined along with the status bits they affect in Table 4.9.

TABLE 4.9 LOGICAL INSTRUCTION SET

		Status Register Bits (x) indicates bits affected by instruction								
Mnemonic	Format	L>	A>	EQ	C	OV	OP	X	INT	MASK
ANDI	(W),#	x	x	x	-	-	-	-	-	- - - -
ORI	(W),#	x	x	x	-	-	-	-	-	- - - -
XOR	G,(W)	x	x	x	-	-	-	-	-	- - - -
INV	G	x	x	x	-	-	-	-	-	- - - -
CLR	G	-	-	-	-	-	-	-	-	- - - -
SETO	G	-	-	-	-	-	-	-	-	- - - -
SOC	G,(G)	x	x	x	-	-	-	-	-	- - - -
SOCB	G,(G)	x	x	x	-	-	x	-	-	- - - -
SZC	G,(G)	x	x	x	-	-	-	-	-	- - - -
SZCB	G,(G)	x	x	x	-	-	x	-	-	- - - -

(ANDI) LOGICAL AND
(XOR) EXCLUSIVE-OR

Logical instructions are primarily used to manipulate the individual bits of an operand. This is opposed to manipulating an entire group of bits as we will learn to do in later sections of this chapter with "shift" instructions. The ANDI instruction utilizes the rule of logic stated:

If A is true and B is true, then C is true.

Specifically the 16 bit value in a Workspace Register is compared bit-by-bit (ANDed) with an immediate value that you specify. If both bits are "true" (that is =1) than the resultant bit is true (set). This procedure is repeated for each bit, with the resultant value obtained being placed in the Workspace Register. For example, if R2 contains a value of >A3D4, the statement:

```
ANDI R2,>6C4E
```

then places the value >2044 in R2 because:

```
>A3D4 = 1010  0011  1101  0100 ANDed
>6C4E = 0100  1100  0100  1110 with this
        ---------------------
>2044 = 0010  0000  0100  0100 results in this
```

Notice how if two "set bits" are compared it results in the setting of the corresponding result bit, however, if the bits do not match the corresponding bit is reset.

Table 4.10 is a "truth table" which gives the result of all possible combinations of zeros and ones that can be "ANDed" together:

TABLE 4.10 LOGICAL AND IMMEDIATE

Workspace Register bit	Immediate operand bit	ANDI result
0	0	0
1	0	0
0	1	0
1	1	1

The logical-or immediate (ORI) instruction compares the 16 bit value in a specified Workspace Register with some immediate value. The logical "OR" utilizes a slightly different version of the previously stated logic rule:

If A is true or B is true, then C is true.

Specifically, the 16-bit value in a Workspace Register is compared bit-by-bit (ORed) with some immediate value that you specify. If either of the two bits being compared is "true" (set to 1), then the resulting bit is also true (set=1). This procedure is repeated for each successive bit with the resulting value being finally placed into the Workspace Register. For example, if R3 holds the value >A3D4, then the statement:

```
ORI  R3,>6C4E
```

places the value >EFDE in R3 because:

```
>A3D4 = 1010 0011 1101 0100  ORed
>6C4E = 0110 1100 0100 1110  with this
>EFDE = 1110 1111 1101 1110  results in this
```

Notice that if either bit being compared is set (=1), then the resultant bit is also set. If neither bit being compared is set, then the resultant bit is reset (=0).

Table 4.11 is a 'TRUTH' table listing the result of all possible combinations of bits that can be ORed together.

TABLE 4.11 LOGICAL OR IMMEDIATE

Workspace Register Bit	Immediate Operand Bit	ORI result
0	0	0
1	0	1
0	1	1
1	1	1

The logical exclusive-or [XOR] utilizes the rule of logic which states:

If either A is true or B is true but not both, then C is true.

The format of the XOR instruction is slightly different then for the ORI and ANDI instructions. The XOR instruction allows the source operand to be specified by any of the general addressing modes while the destination operand must be in a Workspace Register. For example, the statement:

```
XOR   @WORD,R5
```

exclusive-OR's the contents of memory location WORD with the value in R5. The result of this exclusive-OR operation is then placed in R5.

The instruction XOR takes the source operand and does an

exclusive-OR on a bit-by-bit basis with the destination operand. The result of this operation replaces the destination operand. If either of the two bits being compared is "TRUE" (that is =1), but not both, then the resulting bit is also true (set =1). However, if both bits are reset (=0), or both bits are set (=1) then the resulting bit is reset (=0). For example, if R4 contains >A341 and memory location WORD contains >C5F4, then the statement:

```
XOR  @WORD,R5
```

places the value of >66B5 in R5 because:

```
>A341 = 1100 0101 1111 0100 XORed
>C5F4 = 1010 0011 0100 0001 with this
>66B5 = 0110 0110 1011 0101 results in this
```

Notice that if either bit being compared, but not both, is set (=1) then the resulting bit is also set. If neither bit being compared is set then the resulting bit is reset (=0).

Table 4.12 is a 'TRUTH" table listing the result of all possible combinations of bits that can be exclusive-ORed together:

TABLE 4.12 EXCLUSIVE-OR LOGIC TABLE

First Operand Bit	Workspace Register Bit	XOR Result
0	0	0
1	0	1
0	1	1
1	1	0

The value that results from the logical operations ANDI, ORI, and XOR is compared with zero before being placed in the Workspace Register. The results of this comparison then affect the first three bits (L>, A>, EQ) of the Status Register accordingly. For example, if R3 contains >A3D4 then the statement:

```
ORI  R3,>6C4E
```

places the value >EFDE in R3, sets the logical greater than bit of the Status Register while resetting the arithmetic greater than and equal status bits.

The following chart combines all three 'TRUTH' tables. This chart summarizes the effects of the three logical operations:

ANDI	ORI	XOR
0 AND 0 = 0	0 OR 0 = 0	0 XOR 0 = 0
0 AND 1 = 0	0 OR 1 = 1	0 XOR 1 = 1
1 AND 0 = 0	1 OR 0 = 1	1 XOR 0 = 1
1 AND 1 = 1	1 OR 1 = 1	1 XOR 1 = 0

(INV) INVERT

This instruction takes the source operand and reverses all the logic bits. It has the effect of changing each zero in the source operand to one, and changing each one to zero. This is referred to as "taking the one's compliment of a number'. The resulting value is then compared with zero and sets/resets the Status Register accordingly. The new value also replaces the source operand. For example, if R3 contains >3EF4 and memory location >3EF4 contains >A6CC the statement:

```
INV  R3
```

places >C10B in R3 and sets the logical greater than, as well as resetting the equal and arithmetic greater than bits in the Status Register because:

```
>3EF4 = 0011 1110 1111 0100 becomes
        --------------------
>C10B = 1100 0001 0000 1011 on bit-by-bit reversal
```

(CLR) CLEAR WORD

This instruction changes the source operand (16 bit) to zero. That is, all bits are reset. For example, if R6 contains >3001 then the statement:

```
CLR  *R6+
```

clears the contents of memory locations >3000 and >3001 to zero. R6 is then incremented by two (word instruction) so R6 now contains the address >3003. Word operations such as CLR operate on the next lower address when an odd address is specified as the operand, since all memory words have to begin at an even address.

The CLR instruction does not affect the Status Register.

(SETO) SET WORD TO ONE

This instruction is the opposite of CLR in that it replaces the source operand with a full 16-bit word of ones. It does not affect any Status Register bits. For instance, the statement:

```
SETO   @BUFF(R3)
```

places the value >FFFF at the address found by adding R3 to the contents of BUFF. The SETO instruction is useful to signify the end of a file or in the setting up of flag words.

(SOC) SET ONES CORRESPONDING, WORD

This instruction compares two words (16 bits) together. The source operand compares its bits set (1) against the destination operand. All corresponding bits are set in the destination operand regardless of their previous condition. For example, if R3 contains >A3E4 and R4 contains >1C33, then the statement:

```
SOC  R3,R4
```

changes the contents of R4 to >CFF7 because:

```
>A3E4 =  1010  0011  1110  0100  source operand
>1C33 =  1001  1100  0011  0011  destination operand
>CFF7 =  1011  1111  1111  0111  resulting destination
                                 operand
```

This instruction will set the logical greater than bit of the Status Register and reset the equal and arithmetic greater than bits. Notice that the SOC instruction is really an OR operation that can operate on two operands through any general addressing mode.

(SOCB) SET ONES CORRESPONDING, BYTE

This instruction compares the source operand (byte) with the destination operand (byte). It is an OR operation in that if a bit is set in the source operand the corresponding bit is set in the destination operand. The result of this bit-by-bit comparison replaces the destination operand and is then compared with zero. The Status Register bits are set/reset to reflect the results of this comparison. If a word of memory is specified as one of the operands, only the most significant byte (bits 0-7) are OR'ed together. For example, if R3 contains >AA33 and memory location BEST contains >F731, then the instruction:

```
SOCB  R3,@BEST
```

places the value >FB31 at location BEST and sets the logical greater than and odd parity status bits while resetting the arithmetic and equal status bits because:

```
>AA33 =  1011  1010  0011  0011 source operand
>FC31 =  1111  1011  0011  0001 destination operand
>FB31 =  1111  1011  0011  0001 resulting destination operand
```

(SZC) SET ZEROS CORRESPONDING, WORD

This instruction compares the 0's in a source operand (word) with the 0's in a destination operand (word). If a zero bit corresponds then it is not affected. If a zero bit in the source operand corresponds with a one bit in the destination operand, then that bit is reset to zero. The result of this operation is placed in the destination operand. The result is compared with zero and the status bits are set/reset accordingly. For example, if R3 contains >2133 and R4 contains >3399, then the statement:

```
SZC  R3,R4
```

places >2111 in R4 and sets the logical greater than, arithmetic greater than status bits while resetting the equal status bit because:

```
>2133 =  0010  0001  0011  0011 source operand
>3399 =  0011  0011  1001  1001 destination operand
>2111 =  0010  0001  0001  0001 resulting destination operand
```

Notice that if the source operand bit is zero it resets the corresponding destination operand bit. This is a logical OR operation dealing with zeros instead of ones. The opposite of the SOC instruction.

(SZCB) SET ZEROS CORRESPONDING, BYTE

This instruction compares the 0 bits in a source operand (byte) with the 0 bits in a destination operand (byte). If a zero bit corresponds then it is not affected. If a zero bit in the source operand corresponds with a one bit in the destination operand, the destination operand bit is reset to zero. The result of this operation is placed in the destination operand. The resulting binary number from this operation then replaces the destination operand. It is compared with zero and the results of this comparison set/reset the status bits accordingly. For example, if R11 contains the value >2001, location >2001 contains >7D, and location MASK contains >90, then the statement:

```
SZCB  @MASK,*R11
```

results in the contents of memory location >2001 being changed to >11 and the logical greater than, arithmetic greater than status bits being set while the equal bit being reset because:

```
MASK =  1001  0000 source operand
 >7D =  0111  1101 destination operand
 >11 =  0001  0000 resulting destination operand
```

4.8 SHIFT INSTRUCTIONS

Where as logical instructions allow you to manipulate **individual bits, Shift instructions** allow you to manipulate **entire groups of bits.** There are four instructions that allow you to shift the contents of a Workspace Register one or more bit positions to the left or right.

With all four shift instructions the carry status bit (C) holds the value of the last bit shifted out of the register. For example, if a Register is shifted to the right 6 bits, and the sixth bit is a '1', the carry bit in the Status Register is set.

Shift instructions can be divided into two groups; **Logical shift** instructions and **arithmetic shift** instructions. Logical shift instructions displace an operand without regard for its sign. They are used on unsigned numbers and non-numbers such as masks. Arithmetic shift instructions preserve the sign bit. They are used to operate on signed numbers.

All four shift instructions require two operands; a Workspace Register containing a sixteen bit word and a shift count. The count may be any number from 1 to 16.

Table 4.13 outlines the shift instructions and indicates which status bits are affected.

TABLE 4.13 SHIFT INSTRUCTIONS

Mnemonic	Format	(x)	Status Register Bits indicated bits affected by instruction									
			L>	A>	EQ	C	OV	X	INT	MASK		
SRA	(W),#		X	X	X	X	-	-	-	-	-	-
SLA	(W),#		X	X	X	X	X	-	-	-	-	-
SRL	(W),#		X	X	X	X	-	-	-	-	-	-
SRC	(W),#		X	X	X	X	-	-	-	-	-	-

(SRA) SHIFT RIGHT ARITHMETIC
(SLA) SHIFT LEFT ARITHMETIC

These two instructions shift signed numbers. The SRA instruction preserves the sign by replicating the sign bit throughout the shift operation. The SLA instruction on the other hand does not preserve the sign bit, but puts a 1 in the overflow bit of the Status Register if the sign of the number changes after the shift operation. With each bit position shift using SLA, the vacated bit positions are replaced with zeros.

When using shift instructions the first operand is the word to be shifted. The second is the number of bits to be shifted (shift count) which ranges anywhere from 1 to 16. If the shift count in

the instruction is zero, the shift count is taken from Workspace Register R0; bits 12 through 15. If bits 12 through 15 in R0 are all zero then the shift count is 16 bit positions. If a shift count is specified that is greater than 15, then the value is placed in R0 and the least significant four bits are taken as the shift count (bits 12-15). If you specify 0 as the shift count the shift count is 16 bit positions. For example, the statement:

```
SLA  R2,4
```

shifts R2 left four bit positions, if the sign changes the overflow (OV) bit of the Status Register is set.

After a shift takes place, the result is compared with zero and the Status Register bits are set/reset to reflect this comparison. The following are examples of arithmetic shift operations:

1. If R3 contains >12F3 then:

```
SLR  R3,1
```

places a value of >25E6 in R3 and sets the logical greater than and arithmetic greater than status bits while resetting the equal, carry, and overflow status bits because:

```
>12F3 =  0001  0010  1111  0011 R3
         ----  ----  ----  ----
>25E6 =  0010  0101  1110  0110 R3 result (all bits shifted
                                           left 1 bit)
```

2. If R4 contains >FA97 then:

```
SLA  R4,5
```

places a value of >62E0 in R4 and sets the logical greater than, carry, and overflow status bits while resetting the equal status bit because:

```
>FA97 = 1111  1010  1001  0111 R4
        ----  ----  ----  ----
>62E0 = 0101  0010  1110  0000 R4 shifted left 5 bits
```

Note sign change (bit 0), and that the fifth bit shifted out is a one so the carry and overflow bits of the Status Register are set.

3. If R5 contains >6CFD and R0 contains >FFFA then:

```
SLA  R5,0
```

places a value of >F400 in R5 and sets the logical greater than, carry, and overflow bits of the Status Register, while resetting the arithmetic greater than bit because:

```
>6CFD = 0110 1100 1111 1101 R5
        ---- ---- ---- ----
>F400 = 1111 0100 0000 0000 R5 LEFT SHIFT >A BITS.
```

4. If R6 contains >B690 and R0 contains >A3B0 then:

```
SRA  R6,0
```

places a value of >FFFF in R6 and sets the logical greater than, and carry status bits while resetting the arithmetic greater than and equal status bits because:

```
>B690 =  1011  0110  1001  0000 R6
         ----  ----  ----  ----
>FFFF =  1111  1111  1111  1111 R6 right shift 16 bits
                                   sign bit replicated
```

(SRL) SHIFT RIGHT LOGICAL

This instruction shifts unsigned numbers to the right. The vacated bits are filled with zeros. The carry bit of the Status Register holds the value of the last bit shifted out. The shift count is specified in the same manner as with the SRA and SLA instructions, that is if 0 is specified as the shift count, the shift count is taken from bits 12-15 of R0. If these bits equal 0, then the shift count is 16. The result of the shift is placed in the Workspace Register and compared with zero. The Status Register is set/reset to reflect the results of this comparison. The following are some examples of the SRL instructions usage:

1. If R3 contains >FFFF, then the statement:

```
SRL  R3,6
```

places the value >03FF in >R3, sets the logical greater than, arithmetic greater than, and carry status bits while resetting the equal status bit because:

```
>FFFF = 1111  1111  1111  1111 R3
        ----  ----  ----  ----
>03FF = 0000  0011  1111  1111 R3 shifted right 5
                                  bit positions
```

2. If R4 contains >731F, then the statement:

```
SRL  R4,456
```

has a shift count of 8 because:

```
R0 .= 456 = 0000  0001  1100  1000
                              ---- last 4 bits = 8
```

The logical and arithmetic greater than status bits are set, while the equal and carry bits are reset.

(SRC) SHIFT RIGHT CIRCULAR

This instruction shifts the contents of a Workspace Register to the right a specified number of bit positions. The displaced bits are then used to fill the vacated bit positions on the left. The carry status bit contains the value of the bit shifted out of bit position 0 (sign bit with signed numbers). The resulting value is then placed in the Workspace Register. It is compared with zero and the status bits are set/reset to reflect the results of this comparison. For example, if R2 contains >EC62, then the statement:

```
SRC  R2,6
```

results in the value >8BB1 being placed in R2. The logical greater than, and carry status bits are set while the equal and arithmetic greater than bits are reset because:

```
>EC62 = 1110  1100  0110  0010 R2
        ----  ----  ----  ----
>8BB1 = 1000  1011  1011  0001 R2 shifted right 6 bits
```

Note that this instruction fills vacated bit positions with the bits shifted out of position 15. In this example the bit shifted out of bit 0 was one, so the carry bit in the Status Register is set.

There is no "Shift Left Circular" instruction because the same effect can be accomplished with SRC. To shift left a specified count simple shift right a count equal to 16 minus the number. For example, to shift left circular 9 bits use the statement:

```
SRC  R2,16-9
```

or

```
SRC  R2,7
```

The shift instructions also can be used as fast-executing multiply and divide instructions. For instance, shifting the operand one bit position to the left doubles its value (multiplies by 2) and shifting the operand to the right one bit position halves its value (divides by 2).

The following shift instructions show you how to multiply or

divide the contents of a Workspace Register by 4:

```
SRL   R5,2          * DIVIDES UNSIGNED NUMBER BY 4.
SRC   R5,16-2       * MULTIPLIES AN UNSIGNED NUMBER BY 4.
SRA   R5,2          * DIVIDES A SIGNED NUMBER BY 4.
SLA   R5,2          * MULTIPLIES A SIGNED NUMBER BY 4.
```

These shift procedures can save you considerable program execution time when multiplying or dividing numbers. Each shift operation takes a fraction of the time to complete then does a DIV or MPY instruction.

Of course there are limitations, you can only multiply or divide with the shift instructions using multiples of two. You can get around this obstacle by juggling some registers. For example, to multiply the contents of R3 by 10, use the following sequence of instructions:

```
MOV   R3,R4         * Put a copy of R3 in R4.
SCR   R4,16-2       * Shift R4 by 14 (multiply by 4).
A     R3,R4         * Add original R4 (multiply by 5).
SRC   R4,16-1       * Shift R4 by 15 (multiply by 10).
```

This is the same as:

```
[(R3*4)+(R3)]*2=R3*10
```

This instruction sequence involves four steps as opposed to the simple instruction sequence:

```
LI    R4,10
MPY   R3,R4
```

which only requires two steps. However, the former sequence is almost three times faster then the single MPY instruction!

4.9 PSEUDO-INSTRUCTIONS

As mentioned in earlier sections, pseudo-instructions are not really machine language instructions, but rather provide some direction to the assembler as to what to do under certain circumstances. There are two pseudo-instructions that are outlined below in Table 4.14:

TABLE 4.14 PSEUDO-INSTRUCTIONS

Mnemonic	Description
NOP	No operation
RT	Return

(NOP) NO OPERATION

The NOP pseudo-instruction performs no operation when run. It only serves to slow the execution time of the program. No operands are specified and the Status Register is unaffected.

The NOP pseudo-instruction is most often used with the minimemory assembler to allow you to leave "holes" in your code that you may want to come back later and fill with some additional instructions.

(RT) RETURN

This instruction tells your computer to return back to a calling program from a subroutine called up by a BL instruction. For example, the instruction sequence:

```
line#   Label   OP-C   Operands Comments
-----   -----   ----   -----------------
>0001   MAIN    .
                .
                .
>0200           BL     @SUB1    * Branch to location SUB1 and store
>0201   START                   * Return address of next
                .               * Instruction in R11.
                .
                .
                .
>0800   SUB1                    * Beginning of subprogram SUB1.
                .
>0805           RT              * Go back to location START.
```

branches to location SUBP, carries out a sequence of instructions and then returns via the RT to the point just after the BL instruction (in this case we would return to location START).

When the RT instruction is specified the assembler supplies the logic code for the following:

```
B  *R11
```

Remember that when control is transferred by a BL instruction, the link to the calling routine (the Program Counter setting just after the BL instruction) is placed in R11. The RT pseudo-instruction returns control of the program to the instruction following the BL command. Do not alter R11 unless you first save the address somewhere. Do not forget to reload the address in R11 before RT or there is no telling where you will end up!

CHAPTER 4 STUDY EXERCISES

1. Construction an instruction sequence that will multiply a number by 12 using only shift instructions.

2. Where is the return address stored when a BL (Branch & Link) instruction is called?

3. How far can a "jump" be specified in your program?

4. The sole purpose of the Status Register is to provide information on which decisions are based to what group of instructions?

5. What pseudo-instruction is used in combination with the BL instruction?

6. Identify the addressing mode used in each of the following examples:

```
(a)  MOVB R1,*R2              (c)  C   @0A,@VALUE1
(b)  A   *R1+,R2              (d)  MOV  R6,@NUM1+*(R2)
```

5

ASSEMBLER DIRECTIVES

As we have mentioned before, the purpose of the assembler is to convert your source code into the appropriate object code. That is, the assembler program takes your opcodes and their operands, translates them into the appropriate binary numbers, and places them in memory for you. This is the assembly process in its simplest form. By providing some additional commands you can "teach" the assembler program to assist you in creating your assembly program. This is where **Assembler Directives** come in. They are not part of the computers instruction set. They are directions for the assembler to follow during the assembly process. Sometimes they are referred to as "pseudo-instructions" as are NOP & RT, but for now we will put them in a distinct category and refer to them separatly as assembler directives.

There are 28 separate directives that are available, however with the assebler and loader we are using only 22 directives are useful. These will be the ones that we will discuss. The directives can be divided into 5 separate groups based on their functional similarities.

The assembler directives can be divided into the following 5 functional groups:

1. **LOCATION COUNTER DIRECTIVES.** These directives affect the location counter in some way. The location counter is the pointer that determines where the assembler is in the assembly process. It keeps an orderly flow of where data and/or instructions are stored in the memory.

2. **INITIALIZE CONSTANT DIRECTIVES.** These directives let you define symbols. It allows you to assign a symbolic name to an expression. You can also directly define words & bytes.

3. **PROGRAM LINKAGE DIRECTIVES.** These directives allow you to link different assembly program modules together into one long program. This feature can greatly simplify program development.

4. **MISCELLANEOUS DIRECTIVES.** These directives allow you to define extended operations. They also allow you to define the end of your program.

5. **ASSEMBLER OUTPUT DIRECTIVES.** These directives allow you to change the assembler output in order to make it easier to read, such as page lengths, page titles, program identifiers, ect.

LOCATION COUNTER DIRECTIVES

The location counter is the pointer that determines where the assembler stores instructions or data in memory. It sequentially follows the steps in the source listing as it converts it into the object listing. There are 6 useful directives for altering the location counter.

Table 5.0 LOCATION COUNTER DIRECTIVES

Mnemonic	Directive	Format
AORG	Absolute ORiGin	word(expression)
RORG	Relocatable ORiGin	expression
DORG	Dummy ORiGin	expression
BSS	Block Starting with Symbol	word(expression)
BES	Block Ending with Symbol	word(expression)
EVEN	Move to a Word Boundary	

(AORG) ABSOLUTE ORIGIN

When the assembler reaches a AORG directive, the location counter is altered to store the object code for subsequent instructions starting at the location specified by a word. For example if X=7, then the source code statement:

```
LABEL  AORG  >D000+X
```

sets the location counter to >D007, and LABEL is assigned the value >D007.

With the Editor/Assembler you normally let the computer make the placement decisions but AORG gives you the option of making these decisions yourself.

When using the Line-by-Line Assembler with the Mini Memory Module you will use the AORG directive quite frequently to move through various memory locations. See chapter 10 section 10.1 for further details.

(RORG) RELATIVE ORIGIN

You may locate object code **relative** to the current active storage location in memory. The RORG places a value in the location counter which, if encountered in absolute code, also defines succeeding locations as program re-locatable. The dollar sign ($) symbol refers to the current value of the location counter. The statement:

```
LABEL  RORG  $-40
```

overlays the last 20 words (40 bytes) by backing up the location counter 20 words. LABEL is assigned the value that is placed in the location counter.

You may never have occasion to use AORG and RORG in your own programs (provided you are not using the MMM), but you'll encounter them if you ever delve into listings of system programs. For this reason you should know what AORG and RORG do.

(DORG) DUMMY ORIGIN

This directive places a value in the location counter and defines the following address locations as a dummy block or section.

(BSS) BLOCK OF MEMORY STARTING WITH SYMBOL...

The BSS directive allows you to reserve an area of memory for future use. If a label is used it is assigned the location of the first byte in the block. It does this by advancing the location

counter by the value specified in the expression. You reserve memory for use to set up reference tables, arrays, ect. The following code reserves a 32 byte area of memory for your 16 Workspace Registers:

```
MYREG  BSS  32
       .
```

If the AORG directive is to be used in your program, it must precede come before you use any BSS directives.

(BES) BLOCK OF MEMORY ENDING WITH SYMBOL..

This directive is similar to that of the BSS directive in that it reserves a block of memory by advancing the location counter by the value specified in the expression. The label is assigned the location of the last byte in the block. For example, if the location counter contains >200 when the assembler processes the statement:

```
BUFF   BES   >30
```

BUFF is assigned the value >230 and a 48 byte area of memory is reserved. The BES directive can be used to mark the end of a block started with the BSS directive. For example, when the assembler processes the statements:

```
BUFF1  BSS  10
BUFF2  BES  10
```

a 32-byte buffer (memory area) is set beginning at location BUFF1 and ending at location BUFF2.

(EVEN) PUT ON EVEN WORD BOUNDARY

All words in memory begin at an even address. EVEN is a directive that can force the location counter to point to an even address. If the location counter is already at an even address then the directive is ignored, but if the counter is at an odd address, EVEN causes the assembler to jump to the next even address. For example, if the location counter points to address >3001, an EVEN directive makes it point to >3002.

The only time you would need to use the EVEN directive would be to ensure that a statement consisting of only a label is at an even word boundary after a TEXT or BYTE directive.

```
TEXT 'HELLO'
EVEN
DATA >88AF
```

You do not need to use an EVEN directive after a machine instruction or a DATA directive because the assembler automatically advances the location counter to an even address when it processes machine instructions or a DATA directive.

You can avoid much of the hassle of having to use the EVEN directive by simply not specifying a statement consisting of only a label after a TEXT or BYTE directive.

DIRECTIVES THAT INITIALIZE CONSTANTS

These directives allow you to define the values of constants and place the values in bytes or words of memory.

Table 5.1 outlines the directives that initialize constants along with their mnemonics and formats:

TABLE 5.1 DIRECTIVES THAT INITIALIZE CONSTANTS

Mnemonic	Directive	Format
EQU	Define assembly-time constant (EQUate)	expression
DATA	Initialize BYTE	exp,exp...exp
BYTE	Initialize WORD	exp,exp...exp
TEXT	Initialize TEXT	'string'

(EQU) EQUATE— Define constants at assembly time.

This directive assigns a value to some symbol. The label field contains the symbol that you assign. Once you assign the symbol you may use it anywhere you would normally use the expression.

The EQU directive can be used to define a symbol for a 16-bit constant, or another symbolic name. Some examples of the EQU are:

```
JOYX    EQU  >8376    * Constant
UP      EQU  JOYX     * Another symbolic name.
```

You can also specify an index reference through some juggling of the EQU directive like so:

```
MYREG   EQU  >8300    * My own workspace area begins here.
R1HB    EQU  MYREG+2  * Value in high byte of R1 addr. >8302.
R1LB    EQU  MYREG+3  * Value in low byte of R1 addr. >8303.
```

Here we see the individual bytes of a Workspace Register reference through the use of a symbolic equate directive.

(BYTE) INITIALIZE BYTE(S)

This directive can place one or more values in successive bytes of memory. When you specify a label it is assigned the location which the first byte is places at. Each expression is evaluated individually as a signed two's compliment 8-bit number. The following statements show the allowable maximum and minimum values for byte-size variables, in decimal:

```
BUMAX  BYTE  255   * Maximum byte constant, unsigned.
BSMAX  BYTE  127   * Maximum byte constant, signed.
BSMIN  BYTE  -128  * Minimum byte constant, signed.
```

You can also allow the assembler to calculate the value of a constant as in the following example:

```
HERE  BYTE >F+4,-1,-34+>12,>10/8,'A'
```

which initialize five bytes of memory starting with the byte at location HERE. The contents of five successive bytes are >13, >FF, >F0, >02, and >41.

The EVEN directive is often used after the BYTE directive when a DATA or TEXT directive is next in the source code. This is to assure that the next directive begins at an even word boundary.

(DATA) INITIALIZE WORD

This directive only differs from the BYTE directive in that it can place one or more successive values in 16-bit word locations. Each word is evaluated as a signed two's compliment 16-bit number and, if necessary, places a value of >00 in any bytes not filled. The followed statements outline the maximum and minimum values for word-size variables, in decimal:

```
WUMAX  DATA  65535  * Maximum word constant, unsigned.
WSMAX  DATA  32767  * Maximum word constant, signed.
WSMIN  DATA  -32768 * Maximum word constant, signed.
```

Again, it is possible to let the assembler calculate some of the values of the constant as in the following example:

```
HERE  DATA   1+>F3,3121+ C,'AB'
```

which initialize three words of memory beginning at location HERE. The contents of the three successive words are >00F4, >F031, >4142.

The BYTE and DATA directives can be used to set up a data table in memory. To do this simply list the table elements and separate them with a comma. The following sequences of source code set up two 20-element tables, one comprised of bytes and the other

comprised of words:

```
SOUND1  EQU     >34
BTABLE  BYTE    0,0,23,32,43,23,-12,45
        BYTE    36,-120,>3A,'AB',-'DX'          (BYTE TABLE)
        BYTE    64,>A,45,0,3,7*5,SOUND1

SOUND2  EQU     >35
WTABLE  DATA    >3025,>FFAB,-4356,0,23,-34
        DATA    >4567,->35,'VC','G'-4,>5523     (WORD TABLE)
        DATA    >2332,>23,0,5*>34,36,1,34,SOUND2
```

(TEXT) INITILIZE TEXT

The text directive allows you to define a character string as an expression. The string characters are stored in successive bytes of memory as their ASCII hexadecimal eqivalents. The string may be up to 52 characters in length. You may precede the string with a urinary minus (-) sign in which case the last character of the string is negated. When a label is used its location is the first byte in the string. The string must be enclosed in single quotes as shown here with two possible error messages outlined:

```
NICE  TEXT  'THAT NUMBER IS TOO LARGE.
      TEXT  'PLEASE RE-ENTER IT.'
RUDE  TEXT  'TRY IT AGAIN, STUPID'
```

The bytes are filled sequentially by the assembler when processing a TEXT directive. So if the assembler is on an even address when it starts to execute the following directive,

```
MESG  TEXT  'HELLO'
```

the result is >4845, >4C4C, and >4F-- with (--) being determined by the next source statement. For this reason the EVEN directive usually follows a TEXT statement to insure that the next instruction starts at an even word boundary.

PROGRAM LINKAGE DIRECTIVES

Program linkage directives allow you to create programs as separate modules which you later connect together to form one long program. There are a total of five directives that are available to allow you to link programs, however, only three of them can be used with the loader provided with the assembler. These will be the ones we will discuss in depth in the sections that follow.

Table 5.2 outlines the directives that allow you to link programs along with their mnemonics and required formats:

TABLE 5.2 DIRECTIVES THAT LINK PROGRAMS

Mnemonic	Directive	Format
DEF	External DEFinition	symbol,symbol.....symbol
REF	External REFerence	symbol,symbol.....symbol
Copy	Copy	"File Name"

(DEF) EXTERNAL DEFINITION

The DEF directive allows you to makes one or more symbols available to other programs for reference. The DEF directive can be thought of as supplying "entry" points into the program for other programs. The DEF directive must precede the object code that contains the symbols to be defined. For this reason the DEF directive is usually at the beginning of the source code. The following statement shows an example of the usage of the DEF directive:

```
LABEL  DEF  START,SLOAD
```

This statement will cause the assembler to include the symbols START and SLOAD in the object code so that these symbols are available to other porgrams. If a label is specified, it is assigned the current value of the location counter.

(REF) EXTERNAL REFERENCE

The REF directive allows you to have access to one or more symbols defined in other programs. The REF gives you the location of where "entry" into another program is to take place. For instance, the statement:

```
LABEL  REF  START,SLOAD
```

causes the assembler to include the symbols START and SLOAD in the object code so that the corresponding address may be obtained from other programs.

If a symbol is listed in a REF directive inside your program, then the same symbol must be present in the DEF directive of the program that you are trying to link with.

(COPY) COPY FILE

This directive will fetch a file from a diskette during the

separate source file in the assembly process as if it were a series of source statements in the program. The assembler continues right on through. You can use as many COPY directives in a program as you want but if an END directive is encountered the assembly process ends. This happens no matter if the END directive is in the file called up or part of the original program. The following statement is an example of the COPY directives use:

```
LABEL   COPY   "DSK1.GAME1"
        COPY   "DSK1.GAME2"
        COPY   "DSK2.GAME3"
        END
```

This last example will first copy the file GAME1 from disk drive 1 into the computer in order for the assembler to assemble it. It then loads file GAME2 from disk drive one and keeps right on assembling it. Finally, file GAME3 is loaded from disk drive two and it is assembled. The assembler then reaches the END directive and the assembly process stops.

The main use of the COPY directive is to allow you to write programs as separate modules which can then be assembled together. You may want to do this for writing convenience or because the source program is too long to fit on one file.

MISCELLANEOUS DIRECTIVES

The two miscellaneous directives are the Define eXtended Operation directive (XOP) and the END directive. The miscellaneous directives are outlined in Table 5.3 below:

TABLE 5.3 MISCELLANEOUS DIRECTIVES

Mnemonic	Directive	Format
XOP	Define extended OPeration	symbol,term
END	Program END	symbol

(DXOP) DEFINE EXTENDED OPERATION

This directive can only be utilized on the TI-99/4A Home Computer. The DXOP directive will assign a symbol to be used in the operator field to specify an extended operation.

(END) END PROGRAM EXECUTION

The END directive causes the assembly process to stop. The last source statement you put into your program should be an END statement to signify to the assembler that this is where you want the program to end. If you specify a label it is assigned the current value in the location counter.

You can specify and entry point into the program by placing a symbol in the operand field of an END directive. If this is done the program will automatically begin running as soon as it is loaded into the computer. For example, the statement:

```
END  START
```

will cause the program to begin running immediately upon loading starting at address START. If an operand is not specified in the END directive, then you must define the entry point with a DEF directive and type in this entry point in response to the 'PROGRAM NAME' prompt you receive after loading the program using the Editor/Assembler.

If you are using the Line-by-Line Assembler with the Mini Memory Module to program in assembly language the END directive will cause you to exit the assembler. See chapter 10 for more detailed information.

5.4 DIRECTIVES THAT AFFECT ASSEMBLER OUTPUT

There are 5 different directives that you can use to affect assembler output. You may on occasion want to alter the assembler output in order to make the object and/or source code more readily readable.

Table 5.4 outlines the five directives that affect the output of the assembler:

TABLE 5.4 ASSEMBLER OUTPUT DIRECTIVES

Mnemonic	Directive	Format
UNL	DoNot List Source	
LIST	List Source	
PAGE	PAGe Eject	
TITL	page TITLe	'string'
IDT	program IDenTifier	'string'

(LIST) LIST SOURCE
(UNL) DONOT LIST SOURCE

These directives have no effect on the assembler unless you have specified a listing to an output device with the L option of the Editor/Assembler. If you have specified a list file option then the UNL directive will halt the output to the file device such as list file or printer. The UNL directive in not printed out and any source statements following it are not printed.

The LIST directive may be used after a UNL directive to resume printing to an output device such as a list file or printer. The list statement is not printed, but the location counter is incremented and the listing begins with the next source statement.

To summarize the UNL and LIST directives are used to stop and start output by the assembler to a list file device such as a disk drive or printer.

(PAGE) PAGE EJECT

This directive causes the assembler to start printing the source listing (provided the L option has been selected) on a new page. If a label is specified it is assigned the current value of the location counter.

(TITL) PAGE TITLE

The TITL directive will print a heading (provided the L option has been selected) on each subsequent page of the source listing. For example, the statement:

```
TITL 'PROGRAM FOR PRINTING AMORTIZATION SCHEDULE'
```

prints the heading: "PROGRAM FOR PRINTING AMORTIZATION SCHEDULE" on the top of each page of the program listing. The title may be up to 50 characters on length after which the message "OUT OF RANGE" is printed and the title is truncated to the first 50 characters.

(IDT) PROGRAM IDENTIFIER

The IDT directive assigns a name to the program. It is printed in the source listing but serves no other purpose during the assembly process. The name is limited to 8 characters in length after which a "TRUNCATION" error is displayed. If a label in specified it is assigned the value of the current location counter.

CHAPTER 5 STUDY EXERCISES

1. If R1 contains >123A and R2 contains >456C, list the contents of R1 after each of the following statements is executed:

```
(a)  AND   R2,R1
(b)  OR    R3,R1
(c)  XOR   R2,R1
(d)  MOVB  R2,R1
(e)  SLA   R1,2
```

2. What does this sequence do?

```
START MOV  40,R3
      INC  R6
      DEC  R3
      JEQ  OUT
```

3. Write some statements (two lines should suffice) that will store the contents of R3 into a word location called SAVE.

4. What does this instruction do?

```
MPY  >23FF
```

6

UTILITY PROGRAMS

In your computer there exists two distinct areas of random access memory (RAM). The first is termed **CPU RAM (Central Processing Unit RAM)** and is readily manipulated by you. The second is **VDP RAM (Video Display Processor RAM)** and is more difficult to manipulate because it is memory mapped. When you are putting something on the screen, describing sprites, or writing to the sound table you are actually writing to the VDP RAM.

Normally it would be difficult to read and write to the VDP RAM areas because in order to read data you would first have to write a value to a specific address, wait while the data is obtained and then read the data from another address. To write data to VDP RAM the opposite process occurs, namely you place the data in a specific address, write a value to another address to signify that the date is to be written, and then wait while the data is written. This requires an in-depth knowledge of the addresses to use, as well as how to use them.

Fortunately, you have ready access to certain utility programs that allow you to write and read easily to and from the VDP RAM. The following is a listing of utility programs available to you. All utility programs needed by your program must be referenced in a REF statement at the beginning of the source code unless you are using the Mini Memory Module with the line-by-line assembler in which case you should refer to chapter 10.

Table 6.0 outlines the utility programs that are available to you along with a description as to what they do:

TABLE 6.1 UTILITY PROGRAMS

Symbol	Name	Description
VSBW	VDP Single Byte Write	Copies a single byte from CPU RAM into VDP RAM.
VMBW	VDP Multiple Byte Write	Copies Multiple bytes from CPU RAM into VDP RAM.
VSBR	VDP Single Byte Read	Copies a single byte from VDP RAM into CPU RAM.
VMBR	VDP Multiple Byte Read	Copies multiple bytes from VDP RAM into CPU RAM.
VWTR	Write to VDP Register	Copies a single byte from CPU RAM into a VDP register.
KSCAN	Keyboard SCAN	Scans the keyboard and joystick for input and returns it.
GPLLINK	Graphics Programming Language Link.	Links your program to Graphic subroutines that you can use.
DSRLNK	Device Service Routine Link	Links your program to peripheral devices.
XMLLNK	Extended Memory Language Link	Links your assembly program to ROM and RAM routines.

(VSBW) VDP SINGLE BYTE WRITE

This utility allows you to place a single byte in VDP RAM. You place the VDP RAM address you want to write to in R0. You place a copy of the byte you want to write in the most significant byte of R1. You then call the utility. For example, to place >05 at VDP RAM address >0040, you would use the following source code:

```
     REF  VSBW
     .
     .
    LI    R0,>0400
    LI    R1,>0500
    BLWP  @VSBW
```

(VMBW) VDP MULTIPLE BYTE WRITE

This utility program allows you to copy any number of bytes from an area of CPU RAM into an area of VDP RAM. The Block Starting with Symbol (BSS) instruction is usually used to reserve the CPU RAM to hold bytes prior to transfer. To use the VMBW utility, place the VDP RAM address you wish to start writing to in R0. Place the starting address of the information in CPU RAM that you wish to copy in R1. R2 is then loaded with the number of bytes to copy. The utility program is then called. For example, the following source code:

```
        REF   VMBW
        .
        .
BUFFER  BSS   32
        .
        .
        LI    R0,>0300
        LI    R1,BUFFER
        LI    R2,32
        BLWP  @VMBW
```

copies the 32 bytes located in BUFFER into VDP RAM starting at VDP address >0300.

(VSBR) VDP SINGLE BYTE READ

This utility allows you to copy a single byte from an address in VDP RAM into CPU RAM. You do this by placing the VDP address you want a copy of in R0. Then when the utility is called, the value at that address is placed in the most significant byte of R1. For example, if VDP address >0300 contains the value >FF, then the following statements:

```
        REF   VSBR
        .
        .
        LI    R0,>0300
        BLWP  @VSBR
```

places a value of >FF in the most-significant byte of R1.

(VMBR) VDP MULTIPLE BYTE READ

This utility allows you to copy any number of successive bytes from VDP RAM into CPU RAM. Load R0 with the starting address in VDP RAM that you want to start copying from. Load R1 with the CPU address that you want to copy into. You load R2 with the number of bytes to be copied. You then call the VMBR utility.

For example, if you want to copy 40 bytes from VDP RAM beginning at address >0780 into CPU RAM beginning at address BUFFER, you would use the following source code:

```
          REF   VMBW
          .
          .
BUFFER    BSS   >28
          .
          .
          LI    R0,>0780
          LI    R1,@BUFFER
          LI    R2,>28
          BLWP  @VMBW
```

(VWTR) WRITE TO VDP REGISTER

This utility allows you to change the contents of the VDP Workspace Registers. You place the value you want the VDP register to be in the least-significant byte of R0. The most significant byte of R0 is loaded with the VDP register you want to change. For example, the code:

```
          REF     VWTR
          .
          .
          LI      R0,>02CE
          BLWP    @VWTR
```

places a value of >CE in VDP register 2.

NOTE: When changing VDP register 1, place a copy of what you are changing it to at CPU RAM address >83D4. You have to do this because the value at this address is loaded into VDP register R1 when a key is pressed after the screen has "blacked-out" which it does if no key is pressed for a long period of time.

(KSCAN) KEYBOARD SCAN

This utility allows you to check the keyboard and joysticks for input. It also returns the ASCII value of the key that was pressed or the position of a specified joystick. On the next page is Table 6.2 which presents the CPU RAM addresses used by the KSCAN routine.

TABLE 6.2 ADDRESSES USED BY KSCAN UTILITY

Address	Description
>8374	Placing a value here selects the keyboard device to be checked. The following values are allowed: >00 -- Causes entire keyboard to be checked. >01 -- Causes the left side of the keyboard and input from joystick #1 to be checked. >02 -- Causes the right side of the keyboard and input from joystick #2 to be checked.
>8375	This byte holds the ASCII value of the last key pressed. If no key was pressed, then this address contains a value of >FF.
>8376	Holds (Y) position of joystick input.
>8377	Holds (X) position of joystick input.
>837C	Status byte. If a key is pressed then bit 2 is set.

If your program contains a keyboard scanning loop and your program needs to enable interrupts (to move sprites, create sound, ect.) the key scanning loop is an excellent place to do so. The following is an example of how to structure the key scanning loop so that interrupts may be enabled:

```
        REF             * Reference needed utility program.
        .
        .
LOOP    LIMI 2          * Enable interrupts
        LIMI 1          * Disable interrupts
        BLWP @KSCAN     * Call utility program to scan keyboard.
        .
        .
```

A keyboard status byte is located at CPU address >837C. It gives certain status information based on keyboard input. It can be used in combination with a compare ones corresponding (COC) instruction to determine if a key has been pressed. Bit 2 of the status byte is set if a pressed key is detected during execution of the KSCAN utility. The following source listing on the next page can be used to detect a pressed key.

```
          REF  KSCAN          * Reference needed utility.
          .
          .
          .
          .
      SET DATA  >2000         * Binary 0010  0000  0000  0000.
          EQU   >837C         *
          .                   *
          .
GETKEY    BLWP  @KSCAN        * Call up utility program.
          MOV   @STATUS,R3    * Move status word into R3.
          COC   @SET,R3       * Check and see if bit 2 is set.
          JNE   GETKEY        * If no key pressed loop again.
          .
          .
```

An alternative method of checking to see if a key has been pressed is to check address >8375 to see if it contains the value >FF (no key pressed). The following source code performs this check:

```
          REF   KSCAN
            .
            .
KEY       EQU   >8375
HEXFF     BYTE  >FF
            .
            .
GETKEY    BLWP  @KSCAN
          CB    @HEXFF,@KEY * See if a key was pressed.
          JEQ   GETKEY      * If no key pressed, check again.
            .
```

(GPLLNK) GRAPHICS PROGRAMMING LANGUAGE LINK

The following GPL routines can be used by your program to perform some useful tasks such as loading character sets, producing tones, allocating string space ect. All the GPL routines are accessed through GPLLNK. The GPL routines covered in the following sections return to your program after they have finished executing.

In order for you to use the GPLLNK utility you must include the statement REF GPLLNK in your program source code. You must also set the status byte located at address >837C equal to >00 before branching to GPLLNK. The address of the desired GPL routine is put in a DATA statement immediately following the BLWP @GPLLNK instruction. The source code on the following page illustrates these points.

```
REF   GPLLNK      * Reference GPLLNK routine.
.
.
CLR   R1          * R1=0
MOVB  R1,@>837C   * Set Status Register byte=0
BLWP  @GPLLNK     * Call utility.
DATA  >XXXX       * Designate routine desired.
.
```

Table 6.3 lists all the subroutines available with the GPLLNK utility.

TABLE 6.3 GPLLNK UTILITY ROUTINES

Data	Description
>0016	Loads the standard character set into VDP RAM.
>0018	Loads small capitals character set into VDP RAM.
>0020	Executes the "power up" routine.
>0034	Generates the "accept tone".
>0036	Generates the "bad response tone".
>0038	Executes the "get string space" routine.
>003B	Bit reversal routine.
>003D	Cassette device service routine.
>004A	Loads lower case character set into VDP RAM.

The following are complete descriptions of each GPLLNK routine that is available.

DATA >0016 LOAD STANDARD CHARACTER SET

This GPL utility loads the standard set into a designated area of VDP RAM. Before calling this routine, put in CPU RAM address >034A the beginning address in VDP RAM where characters are to be loaded. The following is an example of how to load the standard character set into VDP RAM starting at VDP address >0400:

```
REF    GPLLNK      * Reference needed utility.
 .                 *
 .                 *
LI     R1,>0400    * Beginning address to load characters.
MOV    R1,@>834A   * Place beginning address at >834A.
 .                 *
 .                 *
CLR    R1          * R1=0
MOVB   R1,@>8376   * Move 0 into >837C.
BLWP   @GPLLNK     * Call up utility.
DATA   >0016       * Designate subroutine desired.
 .
```

DATA >0018 LOAD SMALL CAPITALS CHARACTER SET

This GPL routine loads the small capitals character set into a designed area of VDP RAM. Before calling this routine, place the VDP address you want the characters to start loading at CPU RAM address >834A. Use the same source listing as in the previous example except the DATA directive to read DATA >0018.

DATA >0020 EXECUTE POWER-UP ROUTINE

This GPL routine initializes the system. It returns you to the master title screen, clears the VDP circuits and places the default values in the VDP registers, character set, status block, and Color Table. Available VDP RAM size is stored at >8370.

DATA >0034 GENERATE ACCEPT TONE

This routine causes a tone to be generated. It is the same tone that is generated in BASIC in association with a correct input.

DATA >0036 GENERATE BAD RESPONSE TONE

This routine causes a tone to be generated. It is the same tone that is generated in BASIC in response to an incorrect input (error message).

DATA >0038 GET STRING SPACE ROUTINE

This routine sets aside memory space in VDP RAM. CPU address >830C and >830D are loaded with the number of bytes to be reserved. After calling this routine, CPU address >831C points to the beginning of the allocated string space and address >831A points to the first free address in VDP RAM (byte following string). This routine destroys bytes at addresses >8356 through >8359. Addresses starting at 834A onward may also be destroyed in some cases.

DATA >003B BIT REVERSAL ROUTINE

This routine provides a mirror image of a byte. It is most commonly used to from a mirror image of a character or sprite during execution of game programs. Prior to calling this routine, CPU RAM address >834A is loaded with the address of the data in VDP RAM that you want to reverse. Address >834C contains the number of bytes to be reserved.

During execution of this routine, in each byte, bits 0 and 7 are exchanged, bits 1 and 6 are exchanged, bits 2 and 5 are exchanged, and bits 3 and 4 are exchanged. CPU RAM addresses >0830 through >0840 are destroyed.

DATA >003D CASSETTE DSR ROUTINE

This routine allows you to access a cassette recorder. In order for this routine to work a number of condition must be met:

1. The Peripheral Access Block (PAB) and data buffer must be set up in VDP RAM prior to calling the routine.

2. The screen start address must be >00 for prompts issued by the cassette DSR (Device Service Routine).

3. Address >834A is the beginning of the device name (ie. "CS1").

4. Address >8356 points to the first character following the name in PAB.

5. Address >8354 and >8355 are the length of the device name (ie. >0003 for "CS1").

6. The word at address >83D0 should be set to >0000.

7. Address >836D must be set to >08 to indicate a DSR call.

8. The status byte at CPU address >837C must be set to >00.

DATA >004A LOAD LOWER CASE CHARACTER SET

This routine is only available on the TI-99/4A. This routine allows you to load the lower-case character set into a designated area of VDP RAM. Before calling this routine, load CPU RAM address >834A with the starting address in VDP RAM that you want to begin loading the characters.

(DSRLNK) DEVICE SERVICE ROUTINE LINK

This utility allows you to link your assembly language programs with peripheral devices such as printers, disk drives, cassette recorders, ect. It also allows you to link to a subprogram in ROM. Before calling this utility a number of conditions must be set up:

1. A Peripheral Access Block (PAB) must be set up in VDP RAM to describe the characteristics of the device and file to be accessed.

2. The word at CPU RAM address >8356 must be loaded with the value that represents the device or subprogram name length.

3. A DATA directive after the BLWP @DSRLNK is >8 for linkage to a Device Service Routine and >10 for linkage to a ROM routine.

If after the DSRLNK utility is called and no error has occurred, bit (EQ) of the Status Register is reset. If however, and Input/Output error has occurred, the equal bit is set and the error code is stored in the most-significant bit of R0 of the calling programs workspace. Appendix F outlines the Input/Output error codes.

NOTE: You can not use this routine to access a cassette because the cassette Device Service Routine is located in GPL GROM and not normal DSR ROM. In order to access a cassette you must use the statement:

```
BLWP  @GPPLNK
DATA  >003D
```

6.1 (PAB) PERIPHERAL ACCESS BLOCK STRUCTURE

PABs are used by Device Service Routines to access peripheral devices. The structure and format of a PAB is the same for every peripheral. You must place the necessary information describing the peripheral device into the PAB before attempting to open the file.

The PAB is made up of 10 more bytes which provide information to the DSR Utilities regarding the characteristics of the peripheral device and file attributes that you want to access.

Table 6.4 describes the bytes that make up the PAB as well as a description of the information each contains:

TABLE 6.4 PAB STRUCTURE

Byte#	Bits	Contains	Description
0	All	I/O code	I/O code describing current file condition. See following sections for complete description of all allowable I/O codes.
1	-Status Byte-		This byte contains all the information the computer needs to describe the file. It includes information regarding file type, data type, and operation mode. The contents of each bit is outlined below:

TABLE 6.4 PAB STRUCTURE (continued)

Byte#	Bits	Contains	Description
	0-2	Error Code	When an error is detected during an operation the error code is returned here. '00' indicates that no error has been detected. The error codes are further outlined in Table 6.6
	3	Record Style	Place a value of '0' for "Fixed length records" and a value of 'I' for "Variable length records".
	4	Data Format	Place a value of '0' for "DISPLAY" and '1' for "INTERNAL".
	5-6	Operation Mode	"UPDATE"='00', "OUTPUT"='01' "INPUT"='10', "APPEND"-'11'
	7	File Style	Load '0' for "Sequential Files" and '1' for "Relative Files".
2-3	All	Data Buffer Address	This is the address in VDP RAM that you want to put data read from a record or where you place data that you want to write to a record.
4	All	Record Length	The length of each record for "fixed length records" or the value of the maximum length of a "variable length record".
5	All	Character Count	This byte contains the number of characters that you want to WRITE onto a record or it contains the number of characters that is to be READ from a record.
6-7	ALL	RECORD #	This byte is only used with "relative files". It gives the current record number that the next I/O operation is

TABLE 6.4 PAB STRUCTURE (continued)

Byte#	Bits	Contains	Description
			to be performed. But 0 is discarded so that this number can range from a value of 0 through 32767.
8	All	Screen Offset	This byte contains the offset of the screen characters with respect to their normal ASCII values. This is only used with a cassette interface, which requires prompts to be placed on the screen.
9	All	Name Length	This byte contains the length of the File Descriptor begins at byte 10.
10	All	Device/File Descriptor	Contains the device name and if necessary, the file name. The length of this description is given in byte 9.

PAB INPUT/OUTPUT CODES

The following are complete descriptions of each Input/Output code that can be used in Byte 0 of the PAB:

OPEN >00

Before you can do anything with a file or device you must open it. The only exceptions to this are the SAVE and LOAD operations. You cannot alter byte 1 (STATUS BYTE) when an OPEN operation has been performed, the file remains open until a CLOSE operation takes place.

If byte 4 of PAB is set to >0000 (Record Length), the record length that is specified by the attached peripheral is returned in byte 4. If the value for the record length is given by you is greater than 0, then it is used only after being checked against the peripheral in question.

CLOSE >01

This operation will close a previously opened file. If the file was originally opened in APPEND or OUTPUT mode, an END OF FILE (EOF) record is written to the device or file before closing occurs.

After a file is closed you can alter byte 2 (STATUS BYTE) to

change to a new mode of operation before going through the next OPEN operation.

READ >02

This operation will READ a selected record from a designated peripheral device. The obtained information is stored in VDP RAM beginning at the address specified in bytes 2 & 3 (Data Buffer Address) of the PAB. The size of the buffer is number of bytes stored is given in byte 5 (Character Count) of PAB.

When a READ operation takes place, if the length of the inputted record exceeds the buffer size, the remaining bytes are discarded.

WRITE >03

This operation will write to a record from the buffer specified in PAB bytes 2 & 3. The number of bytes that will be written is given in byte 5 of the PAB.

RESTORE/REWIND >04

This operation will reposition the file pointer to the beginning of the file for sequential files. If the file is a relative file, the pointer is set to the record specified in bytes 6 & 7 of PAB.

The RESTORE/REWIND operation can only be carried out if the file was opened in UPDATE or INPUT mode. You can simulate a RESTORE operation when you are using relative files by entering the record at which the file is to be positioned in bytes 6 & 7 (Record #) of the PAB. This will then be the next record accessed in the next operation.

LOAD >05

This operation code will allow you to load the memory image of a file from a peripheral into an area of VDP RAM. You are allowed to use LOAD without a previous OPEN operation.

The following information must be placed in the PAB before instituting a LOAD operation:

1. Place >05 in byte 0 of PAB.
2. Place the starting address in VDP RAM that you want the file to be copied into in bytes 2 & 3 (Data Buffer Address) of the PAB.
3. Place the maximum number of bytes to be loaded in bytes 6 & 7 (Record #) of the PAB.
4. Place the name length in byte 9 of the PAB.
5. Place the file descriptor information in bytes 10 on.

Keep in mind that the LOAD operation will require as much memory space in VDP RAM as the file occupied on a diskette or other medium.

SAVE >06

This operation code will allow you to write a copy of a file in VDP RAM to a peripheral. You are allowed to use SAVE without a previous OPEN operation.

The following information must be placed in the PAB before instituting a SAVE operation.

1. Place >06 in byte 0 of PAB.
2. Place the starting address in VDP RAM from which the file is to be copied in bytes 2 & 3 (Data Buffer Address) of the PAB.
3. Place the number of bytes to be saved in bytes 6 & 7 (Record #) of the PAB.
4. Place the name length in byte 9 of the PAB.
5. Place the file descriptor information starting in byte 10 of PAB.

DELETE FILE >07

This operation code will delete the file specified from the peripheral. A CLOSE operation will then be performed.

DELETE RECORD >08

This operation code will remove a specified record from a relative record file. The number of records that you want to delete is placed in bytes 6 & 7 (Record #) of the PAB. If this operation code is specified with files opened as sequential, an error occurs.

STATUS >09

When the operation code is specified certain status information is returned regarding the peripheral device and file. The status information returned is placed in byte 8 (Screen Offset) of the PAB. Bits 0 through 5 have meaning whether the file is opened or closed, bits 6 & 7 only have meaning when the file is open; otherwise they are reset.

Table 6.5 outlines bits of byte 8 (Screen Offset) and the information regarding status that each returns:

TABLE 6.5 PERIPHERAL STATUS BITS

Bit	Status Information
0	If this bit is set (=1), the file does not exist. If this bit is reset (=0), the file does exist. With devices such as printers this bit would never be set because any file can conceivably exist.
1	The file is write-protected if this bit is set. If reset, this file is not protected and can be written to.
2	Reserved, Always reset.
3	If this bit is set it indicates that the Data Format is **INTERNAL**. If this bit is reset it indicates that the Data Format is **DISPLAY** or that the file is a program file.
4	If this bit is set it indicates that the file is a program file. If this bit is reset it indicates that the file is a data file.
5	If this bit is set it indicates that the record length is VARIABLE. If this bit is reset it indicates that the record length is FIXED.
6	If this bit is set, the file is at the actual physical end of the peripheral and no more data can be written.
7	If this bit is set, the file is at the end of its previously entered data. You can write more data to the file but if you attempt to read past this point an error will be generated.

Now that we have discussed the basic structure of the PAB it is time we go through an example of creating one for your own program so you can better understand how it is accomplished.

Suppose we wanted to OPEN a FIXED 80 file "DSK, FILE1", DISPLAY, INPUT, SEQUENTIAL. To start, byte 0 of the PAB would specify an OPEN operation like so:

```
0000  0000          (OPEN operation code)
```

Byte 1 would indicate FIXED, DISPLAY, INPUT & SEQUENTIAL like so:

```
0000  0100
```

Bytes 2 & 3 would indicate the address in VDP RAM where we will place the data that we will later input to the file. In this case we will put it starting at address >1000 like so:

0001 00000 0000 0000

Byte 4 would indicate our record length, which is 80 or >50:

0101 0000

Byte 5 is our character count which will be:

0000 0000

Bytes 6 & 7 are only used with relative files so we will reset them both to 0 like so:

0000 0000 0000 0000

Byte 8 is our screen offset for a cassette inteface which we are not using, so we reset it to o like so:

0000 0000

The remaining bytes, 10 and on, contain the Device and File Description. Since these are given as ASCII values we will use a TEXT directive to enter it:

```
TEXT   'DSK1.FILE1'
```

Thus, our PAB would look something like this:

```
PAB  EQU  >0004,>1000,>5000,>0000,>000A
     TEXT 'DSK1.FILE!'
```

When accessing files some errors are bound to occur. Errors are returned in bits 0 through 2 of the first byte of the PAB. Table 6.6 on the next page indicates all the possible error codes and their respective meanings.

TABLE 6.6 FILE ACCESS ERROR CODES

Error code	Bits	Meaning
0	000	Bad device name.
1	001	Device is write protected.
2	010	Incorrect file type, incorrect record length, incorrect I/O mode, no records in a relative file.
3	011	Illegal operation; a operation that is not supported on the peripheral or a conflict with the OPEN attribute.
4	100	Out of Buffer space on the device.
5	101	You have attempted to read past the end of the file. The file is closed when this error occurs.
6	110	Device error, bad medium and other hardware problems.
7	111	File error such as data/program file mismatch, non-existent file opened in INPUT mode ect.

NOTE: An error code of 0 indicates that no error has occurred. unless bit 2 of the status byte at address >837C is set. If bit two is set in the Status Register it indicates a bad device name.

Your program should check bits 0 through 1 of byte 1 of the PAB after every I/O operation to see if an error has occurred. You should also clear these bits before every I/O operation.

There are some default values that the DSR will use if no values are specified. The following chart outlines these defaults.

DEFAULT CONDITIONS

1. SEQUENTIAL
2. UPDATE
3. DISPLAY
4. FIXED if relative records, VARIABLE if sequential
5. Record length depends on the peripheral

You also need to construct a PAB in order to comunicate with RS232 interfaces. The following source code illustrates how you may output information to a printer or other peripheral attached via a RS232 interface:

```
000           DEF  START
001           REF  VSBW,VMBW,KSCAN,DSRLNK
002  *
003  MYREG    BSS  >20
004  *
```

```
005  PAB     EQU  >F80
006  STATUS  EQU  >837C
007  PNTR    EQU  >8356
008  PDATA   BYTE 0                * OP-CODE
009          BYTE >10              * Flag status
010          DATA >0002            * VDP buffer
011          BYTE 80               * Record length
012          BYTE 34               * # of characters to write
013          DATA 0                *
014          BYTE 0                *
015          BYTE 12               * Name length
016          TEXT 'RS232.BA=300'   * Device name
017 *
018 ERMSG    TEXT 'ERROR DETECTED= '
019 ERROR#   TEXT '0123456789ABCDEF'
020 *
021 START    LWPI MYREG
022          MOV  R11,R10          * Save return address.
023 *
024 LOOP     BLWP @KSCAN           *
025          MOVB @STATUS,R0       * Key scanning loop
026          JEQ  LOOP             *
027 *
028 STEP1    LI   R0,>0002         *
029          LI   R1,MESS          * Put message on screen
030          LI   R2,34            *
031          BLWP @VMBW            *
032 *
033 STEP2    LI   R0,PAB           *
034          LI   R1,>0300         * Write PAB data to
035          LI   R2,>29           *       VDP RAM
036          BLWP @VMBW            *
037 *
038 STEP3    BL   @STEP4           * Open file
039          LI   R1,>0300         *
040          BLWP @VSBW            *
041          BL   @STEP4           * Write to file
042          LI   R1,>0100         *
043          BLWP @VSBW            *
044          BL   @STEP4           * Close file
045          JMP  LOOP             *
046 *
047 STEP4    LI   R3,PAB+9         * Set
048          MOV  R3,@PNTR         *    PAB ponter
049          BLWP @DSRLNK          *
050          DATA 8                *
051          JEQ  ERROR            *
052          RT                    *
053 *
```

```
054 ERROR   CLR  R4              * Error handling routine
055         MOVB R0,R4           *
056         SWPB R4              *
057         MOVB @ERROR#(R4),R1 * Get error number
058         LI   R0,79           *
059         BLWP @VSBW           * Print
060         LI   R0,62           *     error
061         LI   R1,ERMSG        *          number
062         LI   R2,16           *               and
063         BLWP @VMBW           *                   message
064         B    *R10            *                     on screen
065 *
066 MESS    TEXT 'THIS SENTENCE WILL BE PRINTED OUT!'
067 *
068         END  START
```

CHAPTER 6 STUDY EXERCISES

1. Write a short program that will place the value 34 at VDP RAM address >1000.

2. If CPU RAM address >8375 contains >FF after calling the KSCAN utility, what does that indicate?

3. Write a short program that will select the keyboard device that checks input from the left side of the keyboard and joystick #1.

7

GRAPHICS

Your TI home computer is a versatile machine in that it can construct colorful graphics in a virtual infinite number of different shapes. There are four basic screen modes you can use to aid you in constructing graphics, they are as follows:

1. GRAPHICS MODE
2. MULTICOLOR MODE
3. BIT-MAP MODE
4. TEXT MODE

Before we discuss each individual screen mode and how each can be used, we must first discuss the VDP (Video Display Processor) registers and how they affect what appears on the screen.

7.0 VDP REGISTERS

There are a total of 8 VDP registers labeled 0 through 7. Each register contains a single byte. You can change the contents of a VDP register by using the VWTR utility. The VDP registers contain information that determines how the computer displays graphics on the screen. The following is an example of using the VWTR utility to put a value of >01 in VDP register 7:

```
REF     VWTR          * Reference needed utility program.
.
.
LI      R0, >0701     * VDP R7/value to load=>01
BLWP    @VWTR         * Call utility program
```

The following is a brief description of each VDP register. The default values (values loaded in when the computer is turned on) are also listed:

VDP REGISTER 0

The default for VDP Register 0 is >00 for BASIC, xBASIC, and Editor Assembler.

The following table outlines what each of the bits in VDP Register 0 controls.

TABLE 7.0 VDP REGISTER 0 BITS

Bits	Description
0 - 5	These bits are reserved. All these bits must be reset (=000000).
6	If this bit is set, the screen is put in BIT-MAP MODE.
7	External video enable/disable. Setting this bit enables video input and resetting this bit disables video input.

The default configuration of this register is:

0000 0000

VDP REGISTER 1

The default for VDP Register 1 is >E0 for BASIC, xBASIC, and Editor Assembler.

A copy of VDP Register 1 is located at CPU RAM address >83D4. If no key has been pressed for a long time the computer automatically "blanks" the screen. When subsequently a key is

pressed, the computer reloads VDP register 1 with a copy of what is in address >83D4. Therefore if you want to change VDP register 1, make sure you put a copy of its new value at address >83D4.

Table 7.1 outlines what the bits in VDP Register 1 controls.

TABLE 7.1 VDP REGISTER 1 BITS

Bit	Description
0	Selects 4K or 16K RAM operation. A value of 0 selects 4K RAM operation, and a value of 1 selects 16K RAM operation.
1	Blank enable/disable. Setting this bit (=1) causes the screen to go blank. Resetting this bit (=0) causes the screen to display normally. When the screen is blanked, only the border color remains on it.
2	Interrupt enable/disable. Setting this bit (=1) enables VDP interrupt and a resetting this bit (=0) disables VDP interrupts.
3	If this bit is set, the display is in **TEXT MODE.**
4	If this bit is set, the display is in **MULTICOLOR MODE.**
5	Reserved, must be 0.
6	Sprite size selection. Resetting this bit (=0) selects for standard sized sprites. Setting this bit (=1) selects double-sized sprites.
7	Sprite magnification selection. Setting this bit (=1) selects magnified sprites, and resetting this bit selects unmagnified sprites.

The default configuration for this register is:

$_{\mathcal{C}}$1110 0000

VDP registers 2 through 6 define the beginnings of the Screen Image Table, Color Table, Pattern Descriptor Table, Sprite Attribute Table, and Sprite Descriptor Table. We will discuss each of these tables in great depth in subsequent chapters. But for now it is a good idea not to alter these registers from their default values.

VDP REGISTER 2

The default for this register is >00 in BASIC, XBASIC and Editor Assembler.

This register defines where the Screen Image Table begins The beginning of the Screen Image Table is found by multiplying the value in this register by >400.

VDP REGISTER 3

The default value for this register is >0E in Editor/Assembler, >0C in BASIC and >20 in xBASIC.

This register defines the beginning of the Color Table. The beginning address is found by multiplying the value in this register by >40.

VDP REGISTER 4

The default value for this register is >01 in the Editor/Assembler and >00 in BASIC and xBASIC.

This register defines the beginning of the Pattern Descriptor Table. The beginning address is found by multiplying the contents of this register times >800.

VDP REGISTER 5

The default value for this register is >06 in the Editor/Assembler, BASIC and xBASIC.

This register defines the beginning of the Sprite Attribute Table. The beginning address is found by multiplying the contents of this register times >80.

VDP REGISTER 6

The default value for this register is >00 in the Editor/Assembler BASIC and xBASIC.

This register defines the beginning of the Sprite Description Table. The beginning address is found by multiplying the contents of this register times >800.

VDP REGISTER 7

The default value for this register is >F5 in the Editor/Assembler and >17 in BASIC and xBASIC.

Table 7.2 lists the bits in VDP Register 7 and what each controls:

TABLE 7.2 VDP REGISTER 7 BITS

Bits	Description
0 - 3	Holds the color code for the foreground color in TEXT MODE.
4 - 7	Holds the code for the upper and lower screen border color in all modes.

SUMMARY

The following table summarizes the most important bits in the various VDP registers. These are the bits that you should become familiar with, as a working knowledge of them is necessary in order to program properly.

TABLE 7.3 SUMMARY OF IMPORTANT VDP REGISTER BITS

VDP Register	Bit	Controls
R0	6*	If set, display is in BIT-MAP MODE.
R1	3*	If set, display is in TEXT MODE.
R1	4*	If set, display is in MULTICOLOR MODE.
R1	6	If set, sprites are double-sized.
R1	7	If set, sprites are magnified.

*Resetting these 3 bits puts the display in GRAPHICS MODE.

7.1 GRAPHICS MODE

GRAPHICS MODE is the mode you probably will be programming in most of the time. It allows you to use the standard ASCII characters and define patterns of your own to display on the screen. You can also define the foreground and background colors for any characters. The ASCII character patterns are available to you. You can use sprites and set them in motion in graphics mode.

Graphics consist of characters. Each character is made up by a 8 x 8 dot pattern. The character is defined by turning some dots "on" and leaving others "off" in the pattern.

In order to display a graphic pattern on the screen you have to first describe the shape of the character, then you describe its foreground and background colors, and finally you describe where

on the screen you want the character to be displayed. There are three separate tables that contain the information needed to produce graphics on the screen. The three tables and the information they contain are as follows:

1. PATTERN DESCRIPTOR TABLE
   a) Holds character pattern identifier

2. COLOR TABLE
   a) Holds color code for foreground and background color of character

3. SCREEN IMAGE TABLE
   a) Refers to the screen location of the pattern.

To sum up, graphics are created by setting up information about their shape, color and screen location in the tables. It is recommended that your three graphics tables start at the following VDP RAM addresses (These are the VDP Register default values):

TABLE 7.4 LOCATION OF GRAPHIC TABLES

Table	VDP RAM Table Location
PATTERN DESCRIPTOR TABLE	>0800
COLOR TABLE	>0380
SCREEN IMAGE TABLE	>0000

PATTERN DESCRIPTOR TABLE

The Pattern Descriptor Table can hold up to 256 different patterns or characters. Each character is defined by a "pattern identifier" as outlined in your User's Reference Guide. Each pattern takes up 8 bytes in the Pattern Descriptor Table. Thus character 0 takes up addresses >0800 through >0807, character 1 takes up addresses >0808 through >080F, and character 256 occupies addresses >0FF8 through >0FFF.

In GRAPHICS MODE the standard ASCII character patterns are automatically loaded into the Pattern Descriptor Table by the system. So character 32 (space character) occupies bytes >0900 through >0907, and ASCII character 33 (exclamation point) occupies addresses >0908 through >090F and so on with the other ASCII characters. To find the Table address for any character simply multiply its character number times 8 and add it to >0800. For example to find the table address that starts defining ASCII character 65 (Capital letter 'A'):

[(65) * (8)] + 2048 = 2568 = >0A08

If you want to add additional character patterns of your own but do not want to alter any of the ASCII character patterns already present you can place your own character patterns beginning with character number 128 and extending through 256. Of course, you can alter any pattern in the Pattern Descriptor Table, if you wish.

COLOR TABLE

The Color Table codes for the foreground and background color of each character. Each color code takes up one byte in the Color Table. Each byte codes for the foreground and background color of eight successive characters. The four most-significant bits code for the foreground color and the four least significant bits code for the background color.

The Color Table begins at VDP RAM addresses >0380. The following are the values for the 16 colors available on the TI Home Computer. Note that the values are somewhat different in assembly language then they are in BASIC:

TABLE 7.5 COLOR CODES

COLOR	CODE	BITS SET	COLOR	CODE	BITS SET
Transparent	>0	0000	Light yellow	>8	1000
Black	>1	0001	Light red	>9	1001
Medium green	>2	0010	Dark yellow	>A	1010
Light green	>3	0011	Light yellow	>B	1011
Dark blue	>4	0100	Dark green	>C	1100
Light blue	>5	0101	Magneta	>D	1101
Dark red	>6	0110	Gray	>E	1110
Cyan	>7	0111	White	>F	1111

The byte at address >0380 specifies the colors for characters 0 through 7, the byte at address >0381 specifies the colors for characters 8 through 15, and the byte at address >039F specifies the color of characters 248 through 255.

For example, if we place a value of >F1 at VDP address >0384, characters 32 through 39 are displayed as white on black.

SCREEN IMAGE TABLE

In the BASIC language the screen is divided into 24 rows of 32 columns. A screen location is designated by a row and column number. For example the statement:

```
CALL HCHAR(4,5,65,1)
```

will place the capital letter 'A' in the 4th column row 5.

The computer has no concept of a "screen"; it just views the screen as a series of memory locations. There are no rows and no columns, only 768 possible memory locations numbered 000 through 767. These memory locations begin at VDP address >0000 and extend through address >02FF. These addresses make up the Screen Image Table. Figure 7.6 shows how the consecutive memory locations designate the consecutive screen locations:

FIGURE 7.6 SCREEN IMAGE TABLE/SCREEN POSITION

000	001	002	003	004	.	.	.	.	.	.	.	.	.	029	030	031
032	033	034	035	.	.	.	.	.	.	.	.	.	.	.	062	063
064	965	066	.	.	.	.	.	.	.	.	.	.	.	.	.	095
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
.	.	.												.	.	.
.	.														.	.
.	.														.	.
.	.														.	.
.																.
.																.
.																.
.																.
.																.
736	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	767

If you place the ASCII value of a character in the Screen Image Table, the character will appear in the designated place on the screen. For example, if you place the value 65 in VDP RAM address >23 then the character 'A' will appear in screen position 035. To convert a row and column location into a Screen Image Table address simply use the following formula:

[C + (R * 32)] = P

where C is the column number, R is the row number, and P is the resulting Screen Image Table address.

Now that we know how graphics are put together we can construct a small assembly language program to illustrate how it all goes together. Consider the BASIC program:

```
10  CALL COLOR(1,16,2)
20  CALL HCHAR(4,10,65,1)
30  GOTO 30
```

This short program prints character 65, which is the "A" character, on the screen at row 4 column 10. The character is printed white on a black background. To convert this to an assembly language program we have to load the needed information into the proper tables as demonstrated on the next page.

```
001             DEF   START      * Define program entry point.
002             REF   VSBW       * Reference needed utilities.
003  *
004  MYREG      BSS   >20        * Reserve memory for my registers.
005  *
006  START      LWPI  MYREG      * Pointer to beginning of my
007  *                             workspace.
008             LI    R0,>0384   * Color Table address.
009             LI    R1,>1F00   * Byte to write (white on black).
010             BLWP  @VSBW      *
011  *
012             LI    R0,138     * Screen Image Table address.
013             LI    R1,>4100   * Load character 'A' ASCII 65.
014             BLWP  @VSBW      * Character is displayed in screen
015  *                           position 138.
016  HERE       JMP   HERE       * This holds display on screen.
017             END   START      * Program runs when loaded.
```

Now suppose we want to define a character of our own. In BASIC we would add a CALL CHAR statement to our previous program. We will now define a ball pattern as character 128 and color it red. We will then display it on the screen:

```
10  CALL CHAR(128,"3C7EFFFFFFFF7E3C")
20  CALL COLOR(13,9,1)
30  CALL HCHAR(4,10,128,1)
40  GOTO 40
```

To translate we simply add some additional code to load the new pattern into the Pattern Descriptor Table, and change the color values in the Color Table:

```
001             DEF    START         * Define program entry point.
002             REF    VSBW,VMBW     * Reference needed utilities.
003  *
004    MYREG    BSS    >20
005    BALL     DATA   >3C7E,>FFFF,>FFFF,>7E3C  * Pattern
006    *
007    START    LWPI   MYREG         * Pointer to beginning.
009             LI     R0,>0390      * Load
010             LI     R1,>8000      *     Color
011             BLWP   @VSBW         *           Table (red)
012  *
013             LI     R0,>0C00      * Load ball
014             LI     R1,BALL       * pattern into
015             LI     R2,8          * Pattern Descriptor Table
016             BLWP   @VMBW         *
017  *
018             LI     R0,138        * Screen position
019             LI     R1,>8000      * Character (ball) to write.
020             BLWP   @VSBW         * Place ball on screen.
021  HERE       JMP    HERE          * Hold it on screen.
022             END    START         * Program runs when loaded
```

7.2 MULTICOLOR MODE

MULTICOLOR MODE divides the screen into a series of "boxes". Each box is a 4 x 4 pixels in size. You can define the color of each individual box. There are 64 boxes in a row and there are a total of 48 rows. You are not allowed to define characters or use ASCII characters when in MULTICOLOR MODE. You are allowed to use sprites in MULTICOLOR mode.

To place the screen in MULTICOLOR MODE you must set bit 4 in VDP register 1.

You must place the following values in the Screen Image Table when using MULTICOLOR MODE:

TABLE 7.7 VALUES TO LOAD IN SCREEN IMAGE TABLE

VDP ADDRESSES	VALUES TO LOAD	VDP ADDRESSES	VALUES TO LOAD
>0000 TO >001F	>00 TO >1F	>0180 TO >019F	>60 TO >7F
>0020 TO >003F	>00 TO >1F	>01A0 TO >01BF	>60 TO >7F
>0040 TO >005F	>00 TO >1F	>01C0 TO >01DF	>60 TO >7F
>0060 TO >007F	>00 TO >1F	>01E0 TO >01FF	>60 TO >7F
>0080 TO >009F	>20 TO >2F	>0200 TO >021F	>80 TO >9F
>00A0 TO >00BF	>20 TO >2F	>0220 TO >023F	>80 TO >9F
>00C0 TO >00DF	>20 TO >2F	>0240 TO >025F	>80 TO >9F
>00E0 TO >00FF	>20 TO >2F	>0260 TO >027F	>80 TO >9F
>0100 TO >011F	>40 TO >3F	>0280 TO >029F	>A0 TO >BF
>0120 TO >013F	>40 TO >3F	>02A0 TO >02BF	>A0 TO >BF
>0140 TO >015F	>40 TO >3F	>02C0 TO >02DF	>A0 TO >BF
>0160 TO >017F	>40 TO >3F	>02E0 TO >02FF	>A0 TO >BF

Once you have loaded the Screen Image Table with the above values you can start describing the colors of the boxes on the screen. This is done by placing values in the Pattern Descriptor Table. The Pattern Descriptor Table thus describes colors in MULTICOLOR MODE instead of patterns as it did in GRAPHICS MODE.

The Pattern Descriptor Table should begin at address >0800 in VDP RAM. The first byte in the Pattern Descriptor Table describes the color of the first two adjacent boxes on the first row. The color codes are given on page 103. The left four bits of the byte describe the color of the first box and the right four bits describe the next box on the same row.

The next byte in the table defines the colors of the first two boxes in the second row. The third byte describes the first two boxes in the third row. This continues until the first two boxes

in all 48 rows have been defined. Thus, the first eight bytes in the Pattern Descriptor Table describe the color of the first two columns of boxes. The second group of eight bytes in the table define the colors of the third and fourth columns of boxes. This continues until the last eight bytes in the Pattern Descriptor Table are reached (>0DF8 to >0DFF) which in their turn define the colors of the last two columns of boxes.

7.3 TEXT MODE

In TEXT MODE the screen is 40 columns by 24 rows. You are not allowed to use sprites. Each character is 6 x 8 pixels in size. There are 960 possible screen positions instead of 768. Thus the Screen Image Table is longer. TEXT MODE is most often used in word processing programs.

To place the screen in TEXT MODE you must set bit 3 in VDP register 1. Two colors are available in TEXT MODE, the pixels that are turned off are the color defined in bits 4 through 7 of VDP register 7. The bits that are turned on are the color defined in bits 0 through 3 of VDP register 7.

The tables used in TEXT MODE are set up the same way as the Screen Image Table and Pattern Descriptor Tables are in GRAPHICS MODE except that the Screen Image Table is longer, and in the Pattern Descriptor Table the last two bits of each entry are ignored because each character is only 6 x 8 pixels instead of 8 x 8 pixels as they are in GRAPHICS MODE.

7.4 BIT MAP MODE

BIT-MAP MODE is available only on the TI-99/4A Home computer due to its use of an advanced microprocessor chip. BIT-MAP MODE allows you to define independently each of the 768 screen positions. You can also independently set the color of each pixel in a character. You can use sprites in BIT-MAP MODE but you cannot move them using automatic motion.

In BIT-MAP MODE the Pattern Identifier Codes are stored in the Pattern Descriptor Table. The color codes that describe the colors of these patterns are stored in the Color Table. The Screen Image Table contains the number referencing a given pattern from the Pattern Descriptor Table. The reference numbers range from >00 to >FF each referencing a successive pattern in the Pattern Descriptor Table.

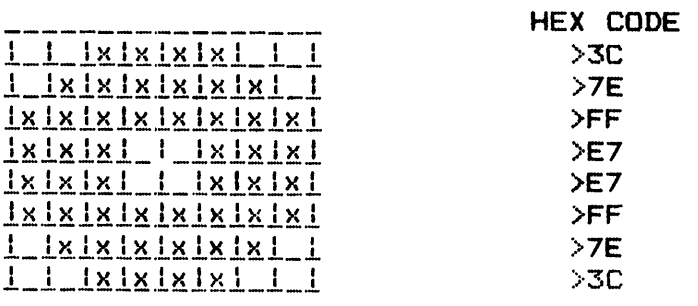
In BIT-MAP MODE you should start the Screen Image Table at VDP RAM address >1800. You do this by setting VDP Register 2 equal to >06. Add the following code to your program to accomplish this:

```
LI      R0,>0206        * (SEE PAGE 80 FOR A REVIEW
BLWP    @VWTR           *     OF THIS UTILITY)
```

The Pattern Descriptor Table begins at VDP RAM address >0000 and is >1800 bytes long. In order to start the table at address >0000 you must load VDP Register 2 with >00 as in the last example. Each pattern identifier code (pattern) takes up 8 bytes in the Pattern Descriptor Table, thus there are 768 possible patterns. See your User's Reference Guide, subprogram CHAR, for further discussion of pattern identifier codes.

The Color Table should begin at VDP RAM address >2000. You can do this by loading a value of >04 into VDP Register 3. The Color Table is >1800 bytes long. Each color code is 8 bytes long. The color codes are described on page 103. The first four bits of each byte code for the color of the pixels that are 'on' in one row of 8 pixels, and the last four bits of each byte code for the color of the pixels that are 'off' in the same row of 8 pixels. For example, the pattern identifier for our ball, "3C7EFFE7E7FF7E3C," which starts at address >0000 of the Pattern Descriptor Table would have >00 as its reference code. You can display the ball anywhere on the screen by entering its reference code in the appropriate place of the Screen Image Table. Other patterns in the table are referenced in the same way. For example, the second group of 8 bytes in the Pattern Descriptor Table (second pattern) are referred to in reference code >01 and so on for all other patterns. The 8 bytes in the Color Table beginning at address >2000 hold the color codes for the ball, the next 8 bytes code for the colors of the next pattern and so on.

Now lets look at an example to illustrate these last points. Say we want the ball to be red with a black background. We also want the ball to have a white square in its center. Our ball pattern would be constructed as follows:



The following code would load this pattern into the Pattern Descriptor Table beginning at VDP address >0000. Don't forget to change the value of VDP Register 4 to >00 first.

```
          .
PATTAB    EQU     >0000
PAT       DATA    >3C7E,>FFFF,>FFFF,>3C7E
          .
          LI      R0,PATTAB
          LI      R1,PAT
          LI      R2,8
          BLWP    @VMBW
```

Now that the pattern is loaded we need to define its colors. First lets draw a map outlinning the colors we want. Black=B, Red=R, and White=W:

```
                    COLOR CODE
                    ----------
|B|B|R|R|R|R|B|B|      >81     *Each row of 8 pixels is coded for
|B|R|R|R|R|R|R|B|      >81     *with one byte.  The first 4 bits
|R|R|R|R|R|R|R|R|      >81     *code for the pixels that are 'ON'
|R|R|R|W|W|R|R|R|      >8F     *in the row, in this case the code
|R|R|R|W|W|R|R|R|      >8F     *is red (8).  The second group of
|B|R|R|R|R|R|R|R|      >81     *bits code for the color of pixels
|B|R|R|R|R|R|R|B|      >81     *that are 'OFF' in the row, in
|B|B|R|R|R|R|B|B|      >81     *this case black (1) or white (F).
```

We can use the following code to load these values into the Color Table beginning at address >2000. Remember to load VDP Register 3 with >04 prior to reaching this segment:

```
        .
        .
COLTAB  EQU   >2000
COLORS  DATA  >8181,>818F,>8F81,>8181
        .
        LI    R0,COLTAB
        LI    R1,COLORS
        LI    R2,8
        BLWP  @VMBW
        .
```

When programming there will be instances when you will want to change which pixels are 'on' and which pixels are 'off' in a character. To do this it will be necessary to calculate the byte and bit position that needs to be changed in the Pattern Descriptor Table. You may also on occasion wish to change the foreground and background colors of a group of eight pixels. To do this it will be necessary to calculate the byte in the Color Table that should be changed.

If you know the X-position and Y-position of a pixel, you can use the following source code to calculate the bit offset and byte that refers to the pixel in the Pattern Descriptor Table. This source listing also provides the byte to change in the Color Table. See page 115 for a description of how how to determine pixel X and Y coordinates.

In this example R0 contains the X-position and R1 contains the Y-position of the pixel:

```
          .
          .
          MOV  R1,R6
          SLA  R6,5
          SOC  R1,R6
          ANDI R6,65287
          MOV  R0,R7
          ANDI R7,7
          A    R0,R6        * R6 is the byte offset
          S    R7,R6        * R7 is the bit offset
          .
```

R6 is the address in the Pattern Descriptor Table that you must change. R7 is the bit that must be altered. The address of the Color Table byte that you will need to change is found by adding >2000 to R6.

The following source code segment can be used to alter the VDP Register values so that the Pattern Descriptor Table, Screen Image Table and the Color Table all begin at the proper addresses required for BIT-MAP MODE:

```
          .
          .
          LI    R0,2       * Put screen
          BLWP  @VWTR      *          in BIT-MAP MODE.
          LI    R0,>0206   * Screen Image Table
          BLWP  @VWTR      *          begins at address >1800
          LI    R0,>0403   * Pattern Descriptor Table
          BLWP  @VWTR      *          begins at address >0000
          LI    R0,>03FF   * Color Table
          BLWP  @VWTR      *          begins at address >2000
          .
          .
```

This next source code segment can be used to initialize the Screen Image Table. The values >00 through >FF are loaded three times in succession:

```
          .
          .
          LI    R0,>1800   *
          CLR   R1         *
          LI    R2,3       *
LOOP      BLWP  @VSBW      *
          INC   R0
          AI    R1,>100    * When  FF+1 is reached, (>00)
          JNE   LOOP       *   no jump is made
          CLR   R1         *
          DEC   R2         * Repeat loading >00 to >FF
          JNE   LOOP       *     three times
          .
```

This final segment can be used to initialize the Color Table. Here we will color all pixels that are "on" black and all pixels that are "off" white. We do this by loading successive values of >F1 into the Color Table:

```
          .
          .
      LI    R0,>2000
      LI    R1,>F100
LOOP  BLWP  @VSBW
      INC   R0
      CI    R0,>3801
      JNE   LOOP
          .
          .
```

The following subprograms illustrate how BIT-MAP MODE can be used. Subprogram INITBM will initialize all tables and place the screen in BIT-MAP MODE. Subprogram TURNON will 'turn-on' a single pixel whose X and Y coordinates have been placed into R3 and R4 respectively. If you are using the Editor/Assembler, you need not type in these subroutines directly into your program. This is because they are all DEF'd. All you need to do is include the subprogram names in a REF statement in your program and follow these steps:

1. Type in the subroutine coding for INITBM and TURNON and save it to disk. Assemble it into an object file named BITMAP.

2. Write your own program which places the X and Y location of the pixel you want to turn-on in R3 and R4 respectively.

3. Include in your program a REF INITBM,TURNON statement. Assemble your program into a file named DEMO (or whatever).

4. Select the LOAD & RUN option and when prompted for the file name type DSK1.DEMO and press ENTER.

5. When prompted for the next file name type DSK1.BITMAP and press ENTER.

6. Press ENTER again.

7. When prompted for a program name, type START and press enter. Program should now execute.

If you are using the Line-by-Line assembler you will have to type in the source code as part of every program that uses BIT-MAP MODE.

This program will draw a rectangle when given the two points of one of its diagonals.

```
001          DEF  START
002          REF  INITBM,TURNON
003 *
004 HIGHX    EQU   65       * Diagonal
005 HIGHY    EQU   50       *        end
006 LOWX     EQU   50       *          points
007 LOWY     EQU   150      *
008 *
009 START    BLWP  @INIT    * Initialize & enter BIT-MAP MODE
010 *
011          LI    R3,HIGHX
012          LI    R4,HIGHY
013 PLOT     BLWP  @TURNON
014          DEC   R3
015          CI    R3,LOWX
016          JNE   PLOT
017          LI    R3,HIGHX
018          INC   R4
019          CI    R4,LOWY
020          JNE   PLOT
021 *
022          LIMI  2
023          JMP   $
024          END
```

The following are the INITBM and TURNON routines:

```
001          DEF   INITBM,TURNON
002          REF   VWTR,VSBW
003 *
004 MYREG    BSS   >20
005 INITBM   DATA  MYREG,$+2
006          LI    R0,2
007          BLWP  @VWTR          * Enter BIT-MAP MODE
008          LI    R0,>0206
009          BLWP  @VWTR          * Screen Image Table = >1800
010          LI    R0,>0403
011          BLWP  @VWTR          * Pattern Descrp. Table = >0000
012          LI    R0,>03FF
013          BLWP  @VWTR          * Color Table = >2000
014 *
015          LI    R0,>1800
016          CLR   R1
017          LI    R2,3
018 LOOP     BLWP  @VSBW
019          INC   R0
020          AI    R1,>100
021          JNE   LOOP
```

```
022             CLR   R1
023             DEC   R2
024             JNE   LOOP
025 *
026             LI    R0,>2000
027             LI    R1,>F100
028 LOOP1       BLWP  @VSBW
029             INC   R0
030             CI    R0,>3801
031             JNE   LOOP1
032 *
033             LI    R0,>1800
034             CLR   R1
035 LOOP2       BLWP  @VSBW
036             DEC   R0
037             JNE   LOOP2
038             RTWP
039 *
049 TURNON      DATA  MYREG,$+2
050             MOV   @6(R13),R3
051             MOV   @8(R13),R4
052             MOV   R4,R5
053             ANDI  R5,7
054             SZC   R5,R4
055             SLA   R4,R5
056             A     R5,R4
057             MOV   R3,R0
058             ANDI  R0,>FFF8
059             S     R0,R3
060             A     R4,R0
061             SWPB  R0
062             MOVB  R0,@>8C02
063             SWPB  R0
064             MOVB  R0,@>8C02
065             NOP
066             MOVB  @>8808,R1
067             SOCB  @GET(R3),R1
068             ORI   R0,>4000
069             SWPB  R0
070             MOVB  R0,@>8C02
071             SWPB  R0
072             MOVB  R0,@>8C02
073             NOP
074             MOVB  R1,@>8C00
075             RTWP
076 GET         DATA  >8040,>2010,>0804,>0201
077             END
```

CHAPTER 7 STUDY EXERCISES

1. Write a few lines of source code that could be used to put the screen in MULTICOLOR MODE.

2. Write a few lines of source code that could be used to display sprites as double-sized and magnified.

3. What will the following source code statements do?

```
REF     VWTR
 .
 .
LI      R0,>0701
BLWP    @VWTR
```

4. Write a complete short program that will display a medium green colored ball-shaped sprite in the center of the screen.

5. How does the computer view the screen?

6. How do you make a program start running immediately upon loading it with the LOAD AND RUN option of the Editor/ Assembler?

8

THOSE SPIRITED SPRITES

Sprites are the mainstay of the game programmer. They can be any shape and color and can occupy screen positions independent of any characters already present. Once set into motion, a sprite can move independently of direct program control. You can magnify or make sprites double-sized. From these characteristics you can see that sprites are a very powerful asset to the programmer intent on designing fast-executing arcade games.

You are allowed to define up to 32 separate sprites on the screen at any given time. You can use sprites in GRAPHICS and MULTICOLOR MODES. You can also use sprites in BIT MAP MODE but you cannot use their automatic motion feature. You cannot use sprites at all in TEXT MODE.

In your computer there are three different tables that collectively contain all the information needed to define sprites. You simply load the desired information into the tables and change it as needed to redefine the characteristics of your sprites. The three tables and the information they contain are as follows:

1. SPRITE ATTRIBUTE TABLE

    a) Sprite position
    b) Sprite color

2. SPRITE DESCRIPTOR TABLE

    a) Sprite pattern identifier
    b) Specify magnified or double-sized sprites

3. SPRITE MOTION TABLE

    a) Define X and Y velocities of sprite

To sum up, sprites are created by setting up information in the three tables that define their position, pattern, color, direction of motion, speed, and their size.

It is recommended that your three sprite tables begin at the following memory locations (default values):

TABLE 8.0 DEFAULT LOCATIONS OF SPRITE TABLES

Table	Table Begins at This VDP Address
SPRITE ATTRIBUTE TABLE	>0300
SPRITE DESCRIPTOR TABLE	>0400
SPRITE MOTION TABLE	>0780

As mentioned before you can have up to 32 separates sprites completely defined and operating at one time. These sprites are numbered from 0 (first sprite) to 31 (last sprite).

Before we discuss the three sprite tables in greater detail we must first understand how the computer defines the screen for sprites. For sprites the computer divides the screen into a series of rows and columns. The columns are labeled starting on the left from 0 to 256 (>00 to >BE). The rows are numbered somewhat differently, starting from the top left, the first row is numbered 256 (>100), followed by the numbers 0 through 255 (>00 to >FF). Each screen location defined by a row and column in this manner is referred to as a pixel. A pixel is the smallest area of the screen that can be turned on or off. Most of the time you

will probably enter the sprite screen position as hexadecimal values, so table 8.1 outlines the rows and columns of all pixel locations in HEX code:

TABLE 8.1 ROW AND COLUMN PIXEL LOCATIONS

	PIXEL COLUMN												
	>00	>01	>02	.	.	.	.	.	.	>FC	>FD	>FE	>FF
>100	.	p1	.	.	.	.	.	.	.	.	.	.	p2
>00	.	.	.	.	.	.	.						
PIXEL ROW >01	.	.	p3	.	.								
>02	.	.	.										
.													
.													
.													
>BB	.	.											
>BD	.	.	.										
>BE	.	p4	.										

Looking at Table 8.2 it can be seen that pixel p1 is in row >100 and column >02, p2 is in row >100 column >FF, p3 is in row >01 column >02, and p4 is in row >BE column >01.

There are some formulas available for converting a graphic row and column location into pixel locations and vice-versa. These formulas are as follows:

TABLE 8.2 GRAPHIC-TO PIXEL INTERCONVERSIONS

GRAPHIC ROW TO PIXEL ROW	GR*8-7=PR
GRAPHIC COLUMN TO PIXEL COLUMN	GC*8-7=PC
PIXEL ROW TO GRAPHIC ROW	INT[(PR+7)/8]=GR
PIXEL COLUMN TO GRAPHIC COLUMN	INT[(PC+7)/8]=GC

GR=graphic row, GC=graphic column, PR=pixel row, PC=pixel column

8.0 SPRITE ATTRIBUTE TABLE

You should begin the Sprite Attribute Table at VDP address >0300. The Sprite Attribute Table holds the information regarding the present screen position of all sprites as well as their colors. The entries in the Sprite Attribute Table change constantly as the position of moving sprites changes.

There are 32 possible sprites numbered 0 through 31. Each sprite takes up four bytes in the Sprite Attribute Table. The first byte is the row or "Y" position of the sprite. The second byte is the column or "X" position of the sprite. The (Y) position starts with >FF then continues with >00, >01, >02 and so on until >BE. The (X) position extends from >00 through >FF. The third byte references the pattern of the sprite as to where it is located in the Sprite Descriptor Table. It can contain any value from >00 to >FF. The fourth byte is the early clock attribute and also codes for the color of the sprite.

When your computer moves sprites it updates the entries in the Sprite Attribute Table. The more sprites it has to update the longer it takes to execute the program. To shorten the time and increase program efficiency you can place a value of >D0 as the Y-location of the lowest numbered non-moving sprite in the Sprite Attribute Table. This indicates that all subsequent sprites are undefined. For example, if you have 10 sprites in motion you should place a value of >D0 at address >0328. If you have no sprites defined, you should place a value of >D0 at address >0300. To sum up, it is recommended that you always let the final unused sprite be undefined by specifying a Y-location of >D0.

The third byte references a pattern in the Sprite Descriptor Table. The pattern reference number can range from >00 to >FF. The value of this byte corresponds to a character defined in the Sprite Descriptor Table. For example, if the third byte contained a value of >80 it would represent the character defined by address >0400 through >0407 in the Sprite Descriptor Table.

The fourth byte controls the early clock of the sprite and its color. The four most significant bits (bits 1-4) control the early clock. If the last bit (bit 4) is reset to zero the early clock is off and the location of the sprite is said to be its upper left-hand corner. This means that the sprite will fade in and out on the right hand side of the screen. If the fourth bit is set to one the early clock is on and the sprites location is shifted 32 pixels to the left. The sprite can then fade in and out on the left side of the screen.

The color of the sprite is determined by the contents of the four least significant bits of the fourth byte in the Sprite Attribute Table. The values are given on the next page.

TABLE 8.3 COLOR CODES

COLOR	CODE	BITS SET	COLOR	CODE	BITS SET
Transparent	0	0000	Medium red	8	1000
Black	1	0001	Light red	9	1001
Medium green	2	0010	Dark yellow	A	1010
Light green	3	0011	Light yellow	B	1011
Dark blue	4	0100	Dark green	C	1100
Light blue	5	0101	Magenta	D	1101
Dark red	6	0110	Gray	E	1110
Cyan	7	0111	White	F	1111

You should take note that the color codes differ slightly in assembly language from their counterparts in BASIC.

The following diagram illustrates how an entry into the Sprite Attribute Table might be constructed. Two sprites are specified.

```
              Sprite 0    Sprite 1

SALIST DATA  >3356,>8001,>A828,>810F,>D0  -- third sprite
              / /   / /                        undefined
             Y X   /  color
               pattern
```

8.1 SPRITE DESCRIPTOR TABLE

The Sprite Descriptor Table describes the patterns of sprites in the same way that the Pattern Descriptor Table describes characters. You will usually begin the Sprite Descriptor Table at address >0400. You can start it at a lower address, but these are usually reserved for the Screen Image Table, Color Table, and Sprite Attribute List. Addresses >0400 through >0407 are defined as sprite pattern >80, sprite pattern >81 occupies addresses >0480 through >040F and so on through sprite pattern >EF which occupies addresses >0778 through >077F.

You can make sprites magnified double-sized or both by writing a value to the two least significant bits of VDP register 1. Table 8.4 which begins on the next page, explains the different sizes and magnifications possible as well as the correct values to write to VDP Register 1.

TABLE 8.4 MAGNIFIED & DOUBLE-SIZED SPRITES

BITS	Description
00	Standard size sprites: Each sprite is 8 x 8 pixels which is the same size as a standard character. HEX (>00)
01	Magnified sprites: sprites is 16 x 16 pixels in size, equal to four standard characters on the screen. Note that the pattern displayed is exactly the same as that for standard size sprites except the sprite is 4x as big. HEX (>01)
10	Double-sized: Each sprite is 16 x 16 pixels on the screen. Each sprite is defined by four consecutive patterns from the Sprite Descriptor Table. For example, if the last two bits (bits 14 & 15) are 01, then if character >80 is referenced the sprite will be formed by characters >80, >81, >82, and >83. The first character, character >80, makes up the upper left hand portion of the sprite, the second character, character >81, makes up the lower left hand portion of the sprite, the third character, character >82, makes up the upper right portion of the sprite, and finally the last character, character >83, makes up the lower right portion of the sprite. HEX (>02)
11	Double-sized magnified sprites: Each sprite is 32 x 32 pixels in size. This is equal to the space occupied by 16 standard size characters on the screen. Sprites are defined in the same way that double-sized sprites are except that each of the four characters is in turn four standard characters in size. HEX (>03)

8.2 SPRITE MOTION TABLE

The Sprite Motion Table specifies the X and Y velocity of each sprite. The Sprite Motion Table begins at address >0780. Before a sprite can be put into motion, several conditions must be met. The first thing that must occur is that your program must allow interrupts. You can enable interrupts with the LIMI 2 instruction however, before your program accesses VDP RAM you will have to disable the interrupts with a LIMI 0 instruction in order that the interrupt handling routine does not alter the VDP write address.

You must also indicate in your program how many sprites will be in motion. This is done by placing a value at address >837A in CPU memory. For example if sprites 2, 5, and 7 are in motion, the number 8 be put in address >837A in order to allow motion of sprites 0, 1, 2, 3, 4, 5, 6, and 7.

A description of the motion of each sprite must be placed in the Sprite Motion Table. Each sprite takes up four bytes in the table. The first byte specifies the (Y) velocity of the sprite, the second byte specifies the (X) velocity of the sprite. The third and fourth bytes are used by the interrupt routine so all you have to do is remember to leave space for them in the table.

The following are allowed as values for (X) and (Y) velocities, also shown are direction of travel:

TABLE 8.5 ALLOWED VALUES FOR X AND Y SPRITE VELOCITIES

Decimal	Hex	Motion	Description
0 to 127	>00 to >7F	Down (Y) Right (X)	Positive velocities. Down or right motion.
-1 to -128	>FF to >80	Up (Y) Left (X)	Negative velocities. Up or left motion.

A value of 1 (>01) will cause the sprite to move one pixel every 16 VDP interrupts. This is approximately once every 16/60ths of a second.

To summarize, in order to put sprites into motion you must:

1. Enable interrupts to occur with the LIMI 2 instruction.
2. The number of sprites in motion must be placed in CPU RAM address >837A.
3. Place descriptions of motion in the Sprite Motion Table which begins at VDP address >0780.

We will now create some programs to illustrate the points covered in this chapter. The first program will place a standard sized sprite in the center of the screen, but we will not put it in motion just yet:

```
001   ********************************************************
002   *                                                      *
003   *   Program to place a red ball-shaped sprite          *
004   *   in the center of the screen.                       *
005   *                                                      *
006   ********************************************************
007           DEF    START
008           REF    VMBW
009   *
010   SATAB   EQU    >0300    *SPRITE ATTRIBUTE TABLE.
```

```
011   SDTAB  EQU  >0400        * SPRITE DESCRIPTOR TABLE.
012   *
013   BALL   DATA >3C7E,>FFFF,>FFFF,>7E3C  * PATTERN CODE.
014   SPAT   DATA >70D0,>8008              * SPRITE ATTRIBUTES.
015          DATA >D000                    * UNDEFINED SPRITE.
016   *
017   MYREG  BSS  >20
019   START  LWPI MYREG
020          LI   R0,SDTAB     * LOAD BALL PATTERN INTO
021          LI   R1,BALL      * SPRITE DESCRIPTOR TABLE.
022          LI   R2,8         *
023          BLWP @VMBW        *
024   *
025          LI   R0,SATAB
026          LI   R1,SPAT
027          LI   R2,8
028          BLWP @VMBW
029   LOOP   JMP  LOOP         * HOLD DISPLAY ON SCREEN.
030          END  START
```

Most programmers think of sprites when referring to moving graphics. Sometimes other methods of imparting motion to characters on the screen are better suited for certain situations. The following program will place six red ball-shaped characters on the screen and scroll the screen upwards moving the characters with it. If you run this program you will notice that the motion of the characters is somewhat jerky, this is because sprites are not used:

```
001   ************************************************************
002   *                                                          *
003   *  Place 6 ball-shaped characters on the screen & scroll   *
004   *  the screen upwards.  This is an example of how to       *
005   *  put graphics into motion without using sprites.         *
006   *                                                          *
007   ************************************************************
008          DEF  GRAPH
009          REF  VSBW,VMBW,VMBR
010   *
011   BALL   DATA >3C7E,>FFFF,>FFFF,>7E3C
012   COLOR  DATA >8100
013   *
014   COLTAB EQU  >0384        *  COLOR TABLE
015   PATTAB EQU  >0908        *  PATTERN DESCRIPTOR TABLE
016   *
017   MYREG  BSS  >20
018   *
019   GRAPH  LWPI MYREG        *
020          LI   R0,COLTAB    *  LOAD FOREGROUND & BACKGROUND
021          MOV  @COLOR,R1    *  COLORS OF BALL CHARACTER INTO
022          BLWP @VSBW        *  COLOR TABLE
```

```
023  *
024           LI    R0,PATTAB    *
025           LI    R1,BALL      *  LOAD THE BALL PATTERN INTO
026           LI    R2,8         *  THE PATTERN DESCRIPTOR TABLE
027           BLWP  @VMBW        *
028  *
029           LI    R0,325       *
030           LI    R1,>2100     *  PLACE 6 BALL SHAPED CHARACTERS ON
031           LI    R2,6         *  THE SCREEN ONE AT A TIME IN
032  LOOP     BLWP  @VSBW        *  DIFFERENT SCREEN POSITIONS
033           AI    R0,33        *
034           DEC   R2           *
035           JGT   LOOP         *  ARE ALL SIX ON SCREEN YET?
036  *
037  LINE1    BSS   >20          *  RESERVE MEMORY TO HOLD SCROLLED
038  LINEX    BSS   >20          *  LINES OF SCREEN
039  *
040  SCROLL   CLR   R0           *  SAVE TOP SCREEN ROW (BEGINNING
041           LI    R1,LINE1     *  WITH POSITION >000) IN LINE1
042           LI    R2,>20       *
043           BLWP  @VMBW        *
044  *
045           LI    R0,>20       *  SAVE SECOND SCREEN ROW IN LINEX
046           LI    R1,LINEX     *
047           LI    R2,>20       *
048           BLWP  @VMBR        *
049  *
050           CLR   R0           *
051  LOOP1    BLWP  @VMBW        *  EACH SCREEN ROW IS SUCCESSIVELY
052           AI    R0,>40       *  READ INTO LINEX AND THEN PRINTED
053           CI    R0,>300      *  IN THE ROW POSITION JUST ABOVE IN
054           JHE   OUT          *  ORDER TO SCROLL THE SCREEN "UP"
055           BLWP  @VMBR        *  WHEN THE LAST ROW IS REACHED
056           AI    R0,>FFE0     *  THE PROGRAM JUMPS TO "OUT"
057           JMP   LOOP1        *
058  *
059  OUT      LI    R0,>2E0      *  PRINT FIRST LINE IN LAST ROW
060           LI    R1,LINE1     *
061           BLWP  @VMBW        *
062  *
063           JMP   SCROLL       *  JUMP BACK TO SCROLL AND REPEAT
064           END   GRAPH        *
```

The source code listing on the next page places our red ball on the screen as a sprite instead of as a graphic. It also places the sprite in motion from left to right across the screen. By pressing any key you can change the magnification of the sprite. The sprite is moved by successively changing its X-location on

the screen. Automatic motion is not used.

```
001  ****************************************************************
002  *                                                              *
003  *                      CALL  SPRITE                            *
005  *  THIS PROGRAM PLACES A RED BALL-SHAPED SPRITE IN             *
006  *  MOTION ACROSS THE SCREEN BY SUCCESSIVELY ALTERING ITS       *
007  *  X-LOCATION. PRESSING ANY KEY ALTERS THE MAGNIFICATION       *
008  *                                                              *
009  ****************************************************************
010          DEF  MOTION
011          REF  VSBW,VMBW,VSBR,VWTR,KSCAN
013  KBOARD  EQU  >8375
014  SKEY    EQU  >8374
015  SATAB   EQU  >0300
016  SDTAB   EQU  >0400
017  *
018  BALL    DATA >3C7E,>FFFF,>FFFF,>7E3C
019  SDATA   DATA >7080,>8008
020          DATA >D000
021  *
022  STATUS  EQU  >837C
023  SET     DATA >2000
024  MYREG   BSS  >20
025  *
026  SPRITE  LWPI MYREG          *
027          CLR  @KEYBOARD     *  KEYBOARD DEVICE=0; SCAN ALL KEYS
028          LI   R0,SDTAB      *  LOAD
029          LI   R1,BALL       *      SPRITE
030          LI   R2,8          *            DESCRIPTOR
031          BLWP @VMBW         *                  TABLE
035  *
036          LI   R0,SATAB      *  LOAD
037          LI   R1,SDATA      *      SPRITE
038          LI   R2,6          *            ATTRIBUTE
039          BLWP @VMBW         *                  TABLE
040  *
041  LOOP    LI   R0,SATAB+1    *
042  READ    BLWP @VSBR         *  GET X POSITION OF SPRITE AND
043          SRL  R1,8          *  SUBTRACT 1 FROM X (X-1)
044          DEC  R1            *
045          JNE  MOVE          *  IF X=0 THEN
046          LI   R1,>00FF      *  LET X=>FF
047  *
048  MOVE    SLA  R1,8          *  WRITE NEW X POSITION
049          BLWP @VSBW         *
050          CLR  R8            *  THIS IS A SHORT DELAY TO
051  DELAY   INC  R8            *  SLOW DOWN THE SPEED OF THE
052          CI   R8,800        *  SPRITE (FOR I=1 TO 800)
053          JNE  DELAY         *
054  *
```

```
056  OUT     BLWP @KSCAN          *
057          MOV  @STATUS,R3      *  CHECK TO SEE IF A KEY HAS
058          COC  @SET,R3         *  BEEN PRESSED
059          JNE  LOOP            *
060  *
061  CHECK   INC  R6              *  R6 IS USED AS A COUNTER TO KEEP
062          CI   R6,4            *  TRACK OF WHICH MAGNIFICATION
063          JLT  GO              *  LEVEL (1 TO 4) WE ARE ON.
064          CLR  R6              *
065  *
066  GO      CI   R6,1            *  SELECT
067          JEQ  MAG             *        NEXT
068          CI   R6,2            *            MAGNIFICATION
069          JEQ  DSIZE           *                        LEVEL
070          CI   R6,3            *
071          JEQ  DSIZEM          *
072  *
073  SMALL   LI   R0,>01E0        *  LOAD R0 WITH THE PROPER VALUE
074          JMP  WRITE           *  TO LOAD INTO VDP REGISTER 1 IN
075  MAG     LI   R0,>01E1        *  ORDER TO CHANGE THE
076          JMP  WRITE           *  MAGNIFICATION
077  DSIZE   LI   R0,>01E2        *
078          JMP  WRITE           *
079  DSIZEM  LI   R0,>01E3        *
080  *
081  ************************************************************
082  *  ACTUALLY LINES 066 THROUGH 079 TAKE UP A GREAT DEAL      *
083  *  OF MEMORY. CAN YOU SUM UP THESE LINES OF CODE INTO       *
084  *  A SIMPLE TWO LINE STATEMENT THAT WOULD WORK AS WELL?     *
085  ************************************************************
086  *
087  WRITE   BLWP @VWTR           *  CHANGE THE VDP REGISTER
088          B    @LOOP
089          END  MOTION
```

This next source code listing again places our red ball on the screen as a sprite. The ball is magnified and is moved using automatic sprite motion. The LIMI 2 instruction is present to allow interrupts to occur. Keep in mind that automatic sprite motion cannot occur without interrupts.

```
001  ************************************************************
003  *                      CALL SPRITE                         *
004  * THIS PROGRAM PLACES A MAGNIFIED SPRITE ON THE SCREEN AND *
005  * PUTS IT IN MOTION USING AUTOMATIC SPRITE MOTION          *
006  ************************************************************
007          DEF  START
009          REF  VMBW,VWTR
010  *
011  NUMB    EQU  >837A
012  SATAB   EQU  >0300
```

```
013  SDTAB   EQU   >0400
014  SMTAB   EQU   >0780
015  BALL    DATA  >3C7E,>FFFF,>FFFF,>7E3C
016  SDATA   DATA  >70D0
017          DATA  >8008
018          DATA  >D000
019  SPEED   DATA  >0505,>0000
020  *
021  MYREG   BSS   >20
022  START   LWPI  MYREG
023          LI    R0,SDTAB       *  LOAD
024          LI    R1,BALL        *      SPRITE
025          LI    R2,8           *             DESCRIPTOR
026          BLWP  @VMBW          *                        TABLE
027  *
028          LI    R0,SATAB       *  LOAD
029          LI    R1,SDATA       *      SPRITE
030          LI    R2,8           *             ATTRIBUTE
031          BLWP  @VMBW          *                      TABLE
032  *
033          LI    R0,SMTAB       *  LOAD
034          LI    R1,SPEED       *      SPRITE
035          LI    R2,4           *             MOTION
036          BLWP  @VMBW          *                   TABLE
037  *
038          LI    R1,1           *  INDICATE NUMBER OF SPRITES IN
039          SLA   R1,8           *  MOTION (1) IN ADDRESS >837A
040          MOVB  R1,@NUMB       *
041  *
042          LIMI  2              *  ENABLE INTERRUPTS
043          JMP   $              *  ENDLESS LOOP TO HOLD DISPLAY ON
044          END   START          *  THE SCREEN
```

9

LET THERE BE SOUND

Both versions of BASIC; BASIC and Extended BASIC- provide a statement that lets you generate sound through the internal console speaker. This statement, CALL SOUND, requires that you specify the duration, frequency and volume of a desired sound.

The frequency can range from 110 Hertz (cycles/sec) to 44,733 Hertz. If you want "noise" instead of a tone to be produced you can specify a negative frequency value of from -1 to -8 depending on the exact noise desired. The duration of a tone or noise can vary from 1 to 4250 milliseconds (.001 to 4.25 seconds). The volume can range from 0 (loudest) to 30 (quietest).

The TI Home Computer is capable of generating up to three tones and one noise simultaneously. Sound is generated using the TMS9919 sound generator controller chip.

In order to produce sound in your assembly language programs a number of conditions must be met. First, you must load the Sound Table with a description of the tone or noise you wish to produce. Secondly, you must set the least significant bit of the byte at CPU address >83FD. This indicates that the Sound Table is in VDP RAM to the computer. Thirdly you must enable interrupts with the LIMI 2 instruction so that sound processing can occur.

The following steps summarize what must be done in order for your program to produce sound:

1. Load the Sound Table which begins at VDP address >83CC with sound data.

2. Set the least significant bit of the byte located at CPU address >83FD to indicate to the computer that the Sound Table is in VDP RAM.

3. Enable interrupts by using the LIMI 2 instruction.

Once all the above conditions are met, you can start the sound generator by placing a value of >01 at CPU address >83CE. This address is used by the interrupt routine as a count-down timer during sound generation.

NOTE: You will have to disable interrupts if you are going to read or write to VDP RAM because the interrupt routine may alter the read/write address. If your program has a key scanning loop this may be a good place to enable/disable your interrupts. See page 81 for an example.

9.0 THE SOUND TABLE

In order to produce sound you must construct a Sound Table that describes the characteristics of the sound you wish to produce. The TI Home Computer has the ability to produce up to three separate tones simultaneously. It can also produce a number of different "noise" sounds. Up to three tones and one noise can be produced simultaneously.

The computer has three tone generators labeled 1, 2, and 3. Noise is produced by a separate noise generator. In order to produce a tone you must enter the following information into the Sound Table:

1. Specify which **TONE GENERATOR** is to produce the tone.
2. Specify the **FREQUENCY** of the tone.
3. Specify the **VOLUME** of the tone.
4. Specify the **DURATION** of the tone.

To produce noise you must enter this information into the Sound Table:

1. Specify **WHITE** or **PERIODIC** noise.
2. Specify **SHIFT RATE** (type of noise).
3. Specify **VOLUME** of noise.
4. Specify the **DURATION** of the noise.

All the bytes that describe the characteristics of a tone or noise except one are referred to as **specification bytes.** The exception is the DURATION byte which is not considered a specification byte.

It takes a total of three specification bytes to hold the generator number, volume and frequency of a tone. Table 9.0 outlines the contents of each of the three bytes. It should be noted now that the frequency is not entered as such (that would be to easy). Instead it is entered as a "**frequency code**" which we will have more on later.

TABLE 9.0 SPECIFICATION BYTES FOR TONES

Byte	Bit#	Holds The following Information:
/	0	This bit is always set (=1).
ONE	1-2	Specifies the Sound Generator.
\	3	This bit is reset (=0).
	4-7	Contains the 4 least significant frequency code bits.
TWO	0-1	These bits are always reset (=00).
\	2-7	Contains the 6 most significant frequency code bits.
/	0	This bit is always set (=1).
THREE	1-2	Indicates Sound Generator used.
\	3	This bit is set (=1).
	4-7	Volume level.

All the noise information requires only two specification bytes. They are structured as outlined in Table 9.1:

TABLE 9.1 SPECIFICATION BYTES FOR NOISE

Byte	Bit#	Holds The Following Information:
	0	This bit is always set (=1).
/	1-2	Specify noise generator (both set =11).
ONE	3	This bit is reset (=0).
\	4	This bit is reset (=0).
	5	Specify WHITE (1) or PERIODIC (0) noise.
	6-7	Indicate TYPE of noise.
/	0	This bit is always set (=1).
TWO	1-2	Indicates Sound Generator used.
\	3	This bit is set (=1).
	4-7	Volume Level.

Bits 1 and 2 in all bytes refer to one of the three tone generators or the noise generator. A bit configuration of 00 selects tone generator #1. A bit configuration of 01 selects tone generator #2. A bit configuration of 10 selects tone generator #3. Finally, a bit configuration of 11 selects the noise generator.

Table 9.2 illustrates several examples of the structure of tone and noise bytes. An X in a bit position is for frequency or volume information that we will cover later.

TABLE 9.2 EXAMPLES OF TONE AND NOISE SPECIFICATION BYTES

Bit configuration	Byte #	Description	HEX
1000 XXXX	1	Tone Generator # 1	>8-
00XX XXXX	2		>--
1001 XXXX	3		>9-
1010 XXXX	1	Tone Generator # 2	>A-
00XX XXXX	2		>--
1011 XXXX	3		>B-
1100 XXXX	1	Tone Generator # 3	>C-
00XX XXXX	2		>--
1101 XXXX	3		>D-
1110 XXXX	1	Noise generator	>E-
00XX XXXX	2		>--
1111 XXXX	3		>F-

FREQUENCY VS. FREQUENCY CODE

You may think that plugging in the desired frequency into the Sound Table is all there is to it. However, it is not that easy. First of all the frequency must be converted into a **frequency code** which is then loaded into the table. The frequency code is defined as half the period of the specified frequency. To save you a lot of time trying to figure out what this means you can use the following formula:

$$\frac{111860.8}{\text{Frequency}} = \text{Frequency Code}$$

Suppose we want to find the frequency code for "middle C" which has a frequency of 523.25 . We simply plug this value into our formula as follows:

$$\frac{111860.8}{523.25} = 213.8$$

We easily find that the proper frequency code equals 213.8, a value that rounds up to 214 (>0D6).

The most significant 6 bits (bits 0-5) of the frequency code are placed in bits 2 through 7 of our second specification byte. The four least significant bits of the frequency code are placed in bits 4 through 7 of our first specification byte. If this sounds a bit confusing don't worry, actually its quite simple. For example, suppose we wanted to define the first two specification bytes of a tone with a frequency of 392 HZ. Further, we want to produce this tone on generator #1. We find from our formula the frequency code which equals 285 or >11D.

1000 XXXX 00XX XXXX = >8---

Here we have selected generator #1. Now we will take our frequency code >11D and place its 4 least significant bits (>D) in bit positions 4 through 7 of our first specification byte:

1000 1101 00XX XXXX = >8D--

Finally, we take the most significant 6 bits of our frequency code (>11) and place them into bit positions 2 through 7 of our second specification byte:

1000 1101 0001 0001 = >8D11

We now have created the first two specification bytes required to produce a tone of 392 HZ on tone generator # 1. The following are some additional examples:

```
1000 0110    0000 1101     [>860D]  Gen #1 freq = 523.25
1010 1110    0000 1011     [>AE0B]  Gen #2 freq = 587.33
1101 1001    0011 1111     [>C93F]  Gen #3 freq = 110.00
```

VOLUME SPECIFICATION BYTE

The third specification byte required for tones holds the volume of the tone. It also holds the value of the generator number you are referring to as did the first specification byte.

The volume is held in bit positions 4 through 7 of the third specification byte for tones. Its value can range from 0 (loudest) to 30 (no sound). When determining the volume level these four bits may be thought of as having a binary zero following them. In this way a volume level of 0001 may be considered as 00010. The following are some examples of the third specification byte:

```
1001 1111      [>9F]    TURNS OFF GENERATOR #1 VOLUME LEVEL = 30
1011 0000      [>B0]    GENERATOR #2, VOLUME LEVEL = 0
1111 0011      [>F3]    NOISE GENERATOR, VOLUME LEVEL = 6
1101 1110      [>DE]    GENERATOR #3, VOLUME LEVEL = 28
```

NOISE SPECIFICATION BYTE

To produce a noise requires only two specification bytes to be loaded into the Sound Table. Referring to Table 9.3 gives the bit values to be loaded into the first specification byte for the desired noise. The second specification byte holds the volume level and is constructed the same way the third specification byte for a tone is constructed except that you specify the noise generator instead of a tone generator.

TABLE 9.3 ALLOWABLE NOISE BIT CONFIGURATIONS

Bit 5	Bits 6 & 7	Description
0	00	"Periodic Noise" Type 1
0	01	"Periodic Noise" Type 2
0	10	"Periodic Noise" Type 3
0	11	"Periodic Noise" varies with the frequency data in tone generator #3
1	00	"White Noise" Type 1
1	01	"White Noise" Type 2
1	10	"White Noise" Type 3
1	11	"White Noise" varies with the frequency data in tone generator #3

Suppose we wanted to construct the two required noise specification bytes for a Type 3 Periodic Noise with a volume level of 6. From Tables 9.1 and 9.3 we put together the first byte like so:

1111 0010 [>F2]

The second specification byte containing the volume information would look like this:

1111 0011 [>F3]

DURATION OF TONE OR NOISE

The DURATION byte is not considered a specification byte. It informs the tone or noise generator how long the tone or noise will last. It is measured in sixtieths (1/60) of a second. Possible values range from 0 (>00) no sound, which stops the generator, to 256 (>FF) which is approximately 4.25 seconds.

LOADING THE SOUND TABLE

One last thing to note before we begin constructing a Sound Table is that when you are setting up a byte table you must indicate the number of specification bytes that you are going to feed to the

sound generator. For example, if you wanted to specify a tone with a frequency of 110 HZ, a volume of 2 and a duration of 0.5 seconds on generator #1, the specification and duration bytes needed are:

>03,>89,>3F,>91,30

The first byte (>03) indicates that there are 3 specification bytes to load into the sound generator. The second and third bytes (>893F) tells us that on generator #1 (>8---) a tone of 110 HZ (>-93F) is desired. The fourth byte (>91) sets the volume level of generator #1 at 2. The last byte (30) specifies a duration of 30/60ths of a second for the tone.

The following are some additional examples of values to load into the Sound Table:

1. >3,>8D,>11,>91,20

```
          -3 specification bytes to load
    1     -Tone Generator #1
 tone     -Frequency = 392.00  FC = >11D
          -Volume level = 2
          -Duration = 20/60ths second
```

2. >3,>A6,>0D,>B5,244

```
          -3 specification bytes to load
    1     -Tone Generator #2
 tone     -Frequency = 523.25  FC = >0D6
          -Volume level = 10    (0101 0)
          -Duration = 244/60ths second
```

3. >9,>83,>15,>A6,>0D,>C7,>09,>91,>B5,>DA,10

```
          -9 specification bytes to load
    3     -Tone Generators #1, #2, & #3
 tones    -Frequencies = 329.63, 523.25 and 739.99
          -Volume levels G1=2, G2=10, G3=20
          -Duration = 10/60ths second
```

4. >2,>E5,>FE,119

```
          -2 specification bytes to load
   1      -Noise Generator (>E0)
noise     -White Noise, Type 2 (>05)
          -Volume level = 28
          -Duration = 119/60ths second
```

5. >1,>9F,0

-This data will terminate the sound in Generator #1.

6. >0B,>8E,>0F,>AD,>17,>CC,>1F,>E3,>90,>B6,>D3,>F6,249

-11 specification bytes to load
-Tone Generators #1, #2, #3 and noise generator
-Frequency = 440.00, 293.66, 220.00
-Periodic Noise of the type that varies with the frequency data loaded into tone generator #3.
-Volume levels G1=0, G2=12, G3=6, NG=12
-Duration = 249/60ths seconds.

The following source code can be used to access the sound controller and start sound processing.

```
        .
        .
SOUNDT  EQU  >1000        * Begin Sound Table at VDP Address >1000
ONE     BYTE >01
        .
START   LI   R10,SOUNDT   *
        .                 * Put VDP address that Sound Table
        MOV  R10,@>83CC   * begins at in CPU address >83CC
        SOCB @ONE,@83FD   * Sound Table is in VDP RAM.
        MOVB @ONE,@>83CE  * Start sound processing.
        LIMI 2
        .
        .
```

The following program plays "HOME ON THE RANGE" on your computer. Note how all three tone generators are used together to produce multiply notes.

```
001 ************************************************************
002 *                                                          *
003 *  Program plays "HOME ON THE RANGE" on your computer.     *
004 *                                                          *
005 ************************************************************
006          DEF   START
007          REF   VMBW
008 *
009 MYREG    BSS   >20
010 SOUNDT   EQU   >1000
011 ONE      BYTE  >01
012          EVEN
013 *
014 START    LWPI  MYREG
015          LI    R0,SOUNDT     *
016          LI    R1,SDATA      *
017          LI    R2,274        *
018          BLWP  @VMBW         *
019 *
020 LOOP1    LIMI  0
021          LI    R10,SOUNDT    *
022          MOV   R10,@>83CC    *
```

```
023           SOCB @ONE,@83FD     *
024           MOVB @ONE,@>83CE    *
027           LIMI 2
029 LOOP2     MOVB  @>83CE,@>83CE    * When CPU address >83CE = 0
030           JEQ   LOOP1            * sound processing is
031           JMP   LOOP2            * finished & program repeats
032 *
033 SDATA     BYTE >03,>8D,>11,>91,40
034           BYTE >04,>AD,>11,>9F,>B1,40
035           BYTE >03,>A6,>0D,>B1,40
036           BYTE >06,>8E,>0B,>AD,>11,>95,>B5,40
037           BYTE >09,>8A,>0A,>A6,>0D,>CD,>11,>95,>B5,>D5,60
038           BYTE >05,>86,>0D,>91,>BF,>DF,20
039           BYTE >03,>82,>0E,>91,40
040           BYTE >03,>8E,>0F,>91,40
041           BYTE >03,>80,>0A,>91,40
042           BYTE >04,>A0,>0A,>9F,>B1,40
043 *
044           BYTE >09,>80,>0A,>A6,>0D,>CD,>10,>95,>B5,>D5,60
045           BYTE >05,>80,>0A,>91,>BF,>DF,20
046           BYTE >03,>80,>0A,>91,20
047           BYTE >03,>8F,>08,>91,40
048           BYTE >09,>8A,>0A,>A6,>0D,>CD,>11,>95,>B5,D5,40
049           BYTE >05,>86,>0D,>91,>BF,>DF,20
050           BYTE >04,>A6,>0D,>9F,>B1,40
051           BYTE >05,>C6,>0D,>9F,>BF,>D1,40
052           BYTE >03,>C2,>0E,>D1,40
053           BYTE >03,>C6,>0D,>D1,40
054           BYTE >03,>CE,>0B,>D1,80
055 *
056           BYTE >03,>CD,>11,D1,40
057           BYTE >04,>8D,>11,>91,>DF,40
058           BYTE >03,>86,>0D,>91,40
059           BYTE >06,>8E,>0B,>AD,>11,>93,>B3,40
060           BYTE >09,>8A,>0A,>A6,>0D,>CD,>11,>95,>B5,>D5,60
061           BYTE >05,>86,>0D,>91,>BF,>DF,20
062           BYTE >03,>82,>0E,>91,40
063           BYTE >03,>8E,>0F,>91,40
064           BYTE >03,>80,>0A,>91,40
065           BYTE >04,>A0,>0A,>9F,>B1,40
066 *
067           BYTE >06,>80,>0A,>AD,>10,>93,>B3,>60
068           BYTE >04,>80,>0A,>91,>BF,20
069           BYTE >04,>A0,>0A,>9F,>B1,40
070           BYTE >09,>8A,>0A,>A6,>0D,>CD,>11,>95,>B5,>D5,50
071           BYTE >05,>8E,>0B,>91,>BF,>DF,>30
072           BYTE >03,>86,>0D,>91,40
073           BYTE >09,>82,>0E,>AD,>11,>CD,>17,>95,>B5,>D5,40
074           BYTE >05,>86,>0D,>91,>BF,>DF,40
075           BYTE >03,>8E,>0B,>91,40
076           BYTE >03,>86,>0D,>91,100
077           BYTE >01,>FF,0
078           END
```

The following table gives you a quick reference guide for frequency specification bytes (specification bytes #1 & #2). Simply look up the desired note or frequency and follow it over to the DATA column to get the first two specification bytes.

The DATA in Table 9.4 always refers to tone generator #1. If you want to produce the tone on generator #2 change the first nybble of the DATA to >A. To produce the tone on generator #3 change the first nybble of the DATA to >C. For example, to produce a tone with a frequency of 5587.65 on generator #2 the DATA would be >A401.

TABLE 9.4 TONE DATA REFERENCE TABLE

NOTE	OCTAVE	FREQUENCY	FREQUENCY CODE	DATA
F	6	5587.65	>014	>8401
E	6	5274.04	>015	>8501
D#	6	4978.03	>016	>8601
D	6	4698.64	>018	>8801
C#	6	4434.92	>019	>8901
C	6	4186.01	>01B	>8B01
B	5	3951.07	>01C	>8C01
A#	5	3729.31	>01E	>8E01
A	5	3520.00	>020	>8002
G#	5	3322.44	>022	>8202
G	5	3135.96	>024	>8802
F#	5	2959.96	>026	>8602
F	5	2793.83	>028	>8802
E	5	2637.02	>02A	>8A02
D#	5	2489.02	>02D	>8D02
D	5	2349.32	>030	>8003
C#	5	2217.46	>032	>8203
C	5	2093.00	>035	>8503
B	4	1975.53	>039	>8903
A#	4	1864.66	>03C	>8C03
A	4	1760.00	>040	>8004
G#	4	1661.22	>043	>8304
G	4	1567.98	>047	>8704
F#	4	1479.98	>04C	>8C04
F	4	1396.91	>050	>8005
E	4	1318.51	>055	>8505
D#	4	1244.51	>05A	>8A05
D	4	1174.66	>05F	>8F05
C#	4	1108.73	>065	>8506
C	4	1046.50	>06B	>8B06
B	3	987.77	>071	>8107
A#	3	932.33	>078	>8807
A	3	880.00	>07F	>8F07
G#	3	830.61	>087	>8708
G	3	783.99	>08F	>8F08

TABLE 9.4 TONE DATA REFERENCE TABLE (Continued)

NOTE	OCTAVE	FREQUENCY	FREQUENCY CODE	DATA
F#	3	739.99	>097	>8709
F	3	698.46	>0A0	>800A
E	3	659.26	>0AA	>8A0A
D#	3	622.25	>0B4	>840B
D	3	587.33	>0BE	>8E0B
C#	3	554.37	>0CA	>8A0C
C	3	523.25	>0D6	>860D
B	2	493.88	>0E2	>820E
A#	2	466.16	>0F0	>800F
A	2	440.00	>0FE	>8E0F
G#	2	415.30	>10D	>8D10
G	2	392.00	>11D	>8D11
F#	2	369.99	>12E	>8E12
F	2	349.23	>140	>8014
E	2	329.63	>153	>8315
D#	2	311.13	>168	>8816
D	2	293.66	>17D	>8D17
C#	2	277.18	>194	>8419
C	2	261.63	>1AC	>8C1A
B	1	246.94	>1C5	>851C
A#	1	233.08	>1E0	>801E
A	1	220.00	>1FC	>8C1F
G#	1	207.65	>21B	>8B21
G	1	196.00	>23B	>8B23
F#	1	185.00	>25D	>8D25
F	1	174.61	>281	>8128
E	1	164.81	>2A7	>872A
D#	1	155.56	>2CF	>8F2C
D	1	146.83	>2FA	>8A2F
C#	1	138.59	>327	>8732
C	1	130.81	>357	>8735
B	0	123.47	>38A	>8A38
A#	0	116.54	>3C0	>803C
A	0	110.00	>3F9	>893F

NOTE: If you need to find a note that is a half-step higher than a given note, you can use the following formula:

(Old Frequency) * 1.059463094 = New Frequency

For example, to find the frequency of a note a half-step higher then Middle 'C':

(523.25) * 1.059463094 = 554.37

10

THE LINE-BY-LINE ASSEMBLER

Although the disc based Editor/Assembler is the most commonly associated package for programming in assembly language, you can also program using the cassette based Line-by-Line assembler in conjunction with the Mini Memory Module. This chapter will attempt to explain the differences in each, as well as how programs written for the Editor/Assembler may be modified for the Line-by-Line assembler.

The first major difference encountered is the fact that the Line-by-Line assembler **assembles** each line of code as soon as it is entered. This is opposed to the disc based Editor/Assembler which assembles the entire source listing at one time after it has been written.

The Line-by-Line assembler provides a 9-page text buffer which allows you to scan previously entered lines of code. You can scroll through the pages of the text buffer by using the up and down arrow keys.

One advantage of learning assembly language on the Line-by-Line assembler is that you get to see what values are placed into memory as soon as a line of source code is entered. This gives you much greater insight into the workings of the computer and how the instructions affect it.

10.0 THE SOURCE CODE STATEMENT

As with the Editor/Assembler each source code statement is made up of four fields. These fields are named and arranged as follows:

LABEL OPCODE OPERAND COMMENT

If you do not specify a LABEL then you must leave a space before typing in the OPCODE. If you use a LABEL the first character must be alphabetic. The second may be any alphanumeric character. The LABEL field when using the Line-by-Line assembler is limited to 2 characters in length. This is our first major difference over the Editor/Assembler which can have LABELS up to 6 characters in length.

The OPCODE, OPERAND and COMMENT fields are all constructed as outlined in section 3.3 of Chapter 3.

10.1 ASSEMBLER DIRECTIVES

There are 7 assembler directives that are recognized by the Line-by-Line assembler. They are:

AORG	Absolute ORigiN
BSS	Block of memory Starting with Symbol
DATA	Word definition (initialization)
END	END program
EQU	Let a LABEL represent a constant
TEXT	String constant definition (initialization)
SYM	Call up SYMbol table

The Directives BSS, DATA, EQU and TEXT are used exactly as outlined in Chapter 5 entitled 'ASSEMBLER DIRECTIVES'. The functions of the remaining directives are outlined in the following sections.

(AORG) ABSOLUTE ORIGIN

You will not need to use this directive much when programming with the Editor/Assembler. However, you will find it indispensable if you attempt to program using the Line-by-Line assembler.

The AORG directive is used to change the value of the Location Counter (which is always an even address). In this way you can jump to any memory location you want in order to alter or review its contents. For example, if you type:

AORG >7D00

the Location Counter will now be set to location >7D00 and the contents of this location will be displayed. If you were to

type in a new source statement and press enter memory location >7D00 would now contain the new value and the Location Counter would advance to address >7D02.

There are basically two main uses for the AORG directive. The first is to point to where you begin entering your program. The second use is to correct errors in the code after you have entered them. To illustrate these two points consider that we are entering the following program where #### represents whatever number happens to be held in a paticular address:

```
Location &
Contents     Instruction      Comments
---------    -------------    ------------------------------------

#### ####        AORG  >7D00  * Go to this address to load program.
7D00 0000  MW    BSS   32     * Reserve my workspace area.
7D20 0201        LWPI  MW     * Put pointer to workspace.
7D22 7D00
7D24 0201        LI    R1,30  * Load a value into R1.
7D26 001E
7D28 0202        LI    R2,64  * Load a value into R2.
7D2A 0040
7D2C 0203        LI    R3,96  * Load a value into R3.
7D2E 0060
7D30 06A0        BL    @S1    * Branch & Link with subprogram S1.
7D32R0000        .
7D34 ####        .
                 .
```

Lets say we have reached this point on entering our program and found that we have made a mistake; instead of loading a value of 30 into R1 we wanted instead to load a value of 32. To get back to address >7D24 and change the value we use the AORG directive as illustrated below:

```
7D34 0000        AORG  >7D24  * Return to address of mistake.
7D24 0201        LI    R1,32  * Insert corrected code.
7D26 0020
7D28 0202        AORG  >7D34  * Go back to where we left off.
7D34 ####         .           * Continue entering program.
                  .
```

(SYM) DISPLAY SYMBOL TABLE

When programming with the Line-by-Line assembler you will specify symbols for operands that have not yet been defined. For example, you may write the instruction JMP S1 where S1 is a destination further along in the program (a point you have not reached to type in yet). The Line-by-Line assembler must keep track of these references somewhere until they are defined by you. These references are kept in a SYMBOL TABLE until you resolve them.

By typing in SYM you can call up the Symbol Table to review references which are unresolved. There are 3 categories within the Symbol Table. These categories and their contents are outlined in Table 10.0.

TABLE 10.0 CATEGORIES OF THE SYMBOL TABLE

Category	Contents
RESOLVED REFERENCES	These are any symbols that have already been defined.
UNRESOLVED REFERENCES (WORD)	These are any symbols that are undefined and are not referenced by a jump instruction.
UNRESOLVED REFERENCES (JUMP)	These are any symbols referenced by a jump instruction.

To see how the SYM directive works lets consider the following example:

```
Location &
Contents          Instruction    Comments
----------        ------------   ----------------------------------
                  AORG  >7D00    * Starting address of program.
7D00  0000  MW    BSS   32       * Reserves workspace area.
7D20  0201        LWPI  MW       * Load pointer to workspace area.
7D22  7D00
7D24  0201        LI    R1,A1    * Load R1 with undefined data.
R0000
7D28  0202  AI    EQU   >0400    * Define A1.
7D26 *0400
7D28  06A0        BL    @S1      * Branch & Link to undefined point.
7DRCR 10FF        JMP   S7       * Jump to undefined destination.
7D2E  ####        SYM            * Now call up Symbol Table.

RESOLVED REFERENCES
MW-7D00  A1-0400

UNRESOLVED REFERENCES (WORD)
S1-7D2A

UNRESOLVED REFERENCES (JUMP)
S7-7D2C

7D2E  ####                       * Ready for next instruction.
```

If a category has no symbols associated with it, that category is not printed on the screen. If all three categories are empty, the SYM directive is erased and the assembler waits for you to

enter the next instruction. A maximum of 32 unresolved references can be displayed by the Symbol Table.

(END) END PROGRAM & EXIT ASSEMBLER

The END directive signifies to the computer that this is the point that your program will end. If you press ENTER after using the END directive you will exit from the assembler. If you press any other key, the END directive is erased and you can keep on entering source code.

After you enter the END directive the statement:

UNRESOLVED REFERENCES

will be displayed on the screen where #### is the number of references that you have not yet resolved. You must go back and figure out which ones they are (by using the SYM directive) and resolve them before attempting to exit from the assembler.

10.2 EDITING

The assembler retains some of the source code in a nine-page buffer which you can review by using the up and down keys to scroll the screen. When the buffer is filled the assembler scrolls back onto the screen to indicate that the buffer is full. Any additional instruction that are entered will overwrite previously written lines in the buffer. Because of this it is a good idea for you to keep a written copy of your source code so that you can refer to it when programming.

Once you start typing a line you cannot "back-up" with the arrow keys to correct a typing error. If you have not pressed "ENTER" you can delete the whole line by pressing "ERASE" and then retyping the entire line correctly. If you have already pressed ENTER then you have to return to that address by way of the AORG directive to change it. If you do not use the label field you can move right to the OPCODE field by simply pressing the SPACE BAR once. You can then move to subsequent fields by pressing the SPACE BAR again.

10.3 ERROR HANDLING

When entering source statements, the Line-by-Line assembler will display an ERROR message under one of three conditions:

1. If you attempt to redefine a previously defined label. For example:

```
                        AORG  >7D00
7D00  0200   MW    BSS   32
7D20  02E0         LWPI  MW
7D22  7D00
7D24  0200   MW    *ERROR*
```

2. If you attempt to enter an undefined opcode or directive. For example:

```
7D00  0200   MW    BSS   32
7D20  02E0         LWPP  *ERROR*
```

3. If you attempt to exceed the reach (256 bytes) of a jump instruction. For example:

```
7D00  ####     JEQ  JI
   .
   .
7E02  #### JI CLR  R1
7D00  *R-ERROR*
```

NOTE: If you even suspect that a jump instruction to an as yet undefined label might possibly be out of range (that is more than 256 bytes away) you would be better off using a B (branch) instruction. If you did not you couldn't go back later because a Branch requires 4 bytes of memory while a jump instruction requires only 2. The following illustrates these points:

```
THIS WAY                      NOT THIS WAY
--------                      ------------
7D00  ####  JNE$+6            7D00  ####  JEQ  JI
7D02  0460  B  @JI            .
7D04  7E02                    .
7D06  C081  MOV R1,R6         .
  .                           .
7E02  ####  JI  CLR R1        7E02  ####  JI CLR R1
```

10.4 THE REFERENCE/DEFINITION TABLE

Once you have finished entering your program you must also enter the program name and location of its starting point in the REF/DEF table so that mini memory module can find it.

The following is a short program that will print a message on the screen. We will then demonstrate how to use assembler directives to enter its name and starting point in the REF/DEF table:

```
                    AORG  >7D00
7D00  ####  WS      BSS   32
7D20  ####  MW      EQU   >6028           * EQUATE VMBW UTILITY.
7D20  484F  A1      TEXT  'HOW ARE YOU?'  * MESSAGE TO DISPLAY.
7D22  5720
7D24  4152
7D26  4520
7D28  594F
7D2A  5535
7D2C  02E0  ST      LWPI  WS              * POINTER TO WORKSPACE AREA.
7D2E  6028
7D30  0200          LI    R0,138          * SCREEN TABLE LOCATION.
7D32  008A
7D34  0201          LI    R1,A1           * BEGINNING OF MESSAGE.
7D36  7D20
7D38  0202          LI    R2,12           * # OF BYTES TO WRITE.
7D3A  000C
7D3C  0420          BLWP  @MW             * BRANCH TO VMBW UTILITY.
7D3E  6028
7D40R10FF           JMP   $               * HOLD DISPLAY ON SCREEN.
7D40*10FF
7D42                END
```

Assuming that you have just entered the preceding program exactly as written and have not exited from the assembler, the screen will appear as follows:

```
7D42  ####  END
      0000  UNDEFINED REFERENCES
```

Do not press ENTER at this point (if you do you will exit from the assembler). Instead you should enter the following code to place the program name and starting location in the REF/DEF table so that you may run the program:

```
7D42  ####  AORG  >701C
```

>7D42 is the first address that is not used in your program. That is, it is the First Free Address in the Module (FFAM). #### represents whatever value happens to be contained in address >7D42. Address >701C holds the FFAM.

```
701C  ####
```

represents the address of the old FFAM. We need to put the new FFAM (>7D42) here.

```
701C  7D42  DATA  >7D42
```
Remember, FFAM is the First Free Address that follows your program, in this case >7D42.

```
701E  7FE8
```
Address >701E holds the Last Free Address in the Module (LFAM). Subtract this value from the FFAM; if the difference is 7 bytes or more, you have enough room to insert your program name.

```
701E  7FE0  DATA  >7FE0
```
Subtract 8 bytes from the old LFAM and place the result at address >701E like we have done here by using the DATA directive.

```
7020  ####
```
Location counter advances to here displaying any data located at this address. We now need to jump to the REF/DEF table and enter our program name.

```
7020  #### AORG  >7FE0
```
Jump to new entry point in REF/DEF table. >7FE0 #### Data at this address is displayed.

```
7FE0  5052 TEXT  'PRINT1'
7FE2  494E
7FE4  5431
```
Enter the program name as PRINT1. The program name must be exactly 6 characters long. The characters making up the name are stored in six bytes beginning at location >7FE0.

```
7FE6  ####
```
Location counter advances to this next location, where we will define the 2-character entry point into our program.

```
7FE6  7D2E  DATA  ST
```
Entry point at where we want program to start running.

```
7FE8  ####  END
```
Enter the END directive and press ENTER to leave the assembler.

We can now run this last program by selecting the RUN option from the MINI MEMORY selection list and typing in PRINT1 for the PROGRAM NAME? prompt and pressing ENTER.

To summarize, in order to run your assembly program you must:

1. Place new FFAM at address >701C.

2. Compare new FFAM with LFAM to see if there is a difference of 7 bytes or more. If there is then you can proceed.

3. Subtract 8 bytes from old LFAM and place the resulting value at address <701E with a DATA directive,

4. Jump to new LFAM and by using a TEXT directive enter your program name which must be exactly 6 characters in length.

5. Define the entry label into your program with a DATA directive at address LFAM+6.

If you have a disk memory system, you can use the LOAD AND RUN option of the MINI MEMORY module to execute assembly programs that were written using the Editor/Assembler system. When the mini-memory comes across a BLWP @VMBW instruction while it is loading from a disk system, it will look up the address it needs in order to use the required utility. It will do this with all subsequent utilities it encounters.

Thus, even though you can not create a program with the line-by-line assembler using the instruction BLWP @VMBW you can RUN programs that contain these symbols with the mini-memory module when the LOAD AND RUN option is used. All predefined symbols in the Editor/Assembler will load correctly into the Mini-Memory Module because they are all predefined in an internal table used by the loader.

10.5 SAVING PROGRAMS

You can save your assembly language program on cassette tape in the following manner:

1. Select EASY BUG option from the selection menu.

2. Use the S command.

3. You can enter the actual starting and ending address of your program, but it is recommended that you enter a starting address of >7000 and an ending address of >7FFF in order to include the REF/DEF table and pointers. If you do not do this you will have to re-enter the program name in the REF/DEF table every time you load the program.

10.6 UTILITY PROGRAMS

All the utility programs discussed in chapter 6 are available when using the Line-by-Line assembler. However the Line-by-Line assembler does not recognize the predefined symbols that the Editor/Assembler package does. With the Line-by-line assembler you simply cannot reference the needed utilities, you have to branch directly to the address the utility is located at. The following routine is an example of how utility programs are accessed when programming with the Line-by-Line assembler.

```
Location
& Contents      Instruction          Comments
----------      -----------          ----------------------
                AORG >7D00
7D00 ####  MW   BSS  32              *
7D20 02E0       LWPI MW              *
7D22 7D00
7D24 ####  GP   EQU  >6018          * GPLLNK begins @>6018
7D24 04C1       CLR  R1              * Set status byte=0
7D26 D801       MOVB R1,@>837C       *
7D28 837C                            *
7D2A 042A       BLWP @GP             * BL with GPLLNK
7D2C 6018
7D2E 0034       DATA >0034           * Accept tone routine
7D30 ####       END                  * Exit assembler
```

This short program uses an equate directive to create a symbol (GP) for the GPLLNK utility which begins at address >6018. Of course, the program could have just as easily referenced the address directly. The following table lists the available ROM utilities and their respective addresses.

TABLE 10.1 ROM UTILITY LOCATIONS

Address	E/A Symbol	Utility
>6018	GPLLNK	Link to GROM routine
>601C	XMLLNK	Link to ROM routine
>6020	KSCAN	Keyboard scan routine
>6024	VSBW	VDP single byte write
>6028	VMBW	VDP multiple byte write
>602C	VSBR	VDP single byte read
>6030	VMBR	VDP multiple byte read
>6034	VWTR	Write to VDP Register
>6038	DSRLNK	Device service routine link
>603C	LOADER	Link to tagged object loader
>6040	NUMASG	Numeric assignment routine
>6044	NUMREF	Get numeric parameter
>6048	STRASG	String assignment routine
>604C	STRREF	Get string parameter
>6050	ERR	Error reporting routine
>6F0E		Beginning of REF/DEF Table
>6FFF		End of REF/DEF Table

11

CONVERTING BASIC TO ASSEMBLY LANGUAGE

Using a high level language such as BASIC or xBASIC to create a program is relatively easy. The sprite capabilities and the clear straight-forward instruction set give you a great deal of control during program construction.

In fact, in most applications BASIC is ideally suited over most other languages for programming. However, when fast-executing arcade style games or other similarly designed programs are needed, BASIC can be intolerable slow. To overcome this speed barrier, we must deal on a level much closer to the level the computer actually communicates on. That is why we write this type of program in assembly language. Assembly language executes at many times the speed of BASIC. Unfortunately, assembly language for many people is much more difficult to work with. One way to circumvent this difficulty is to first write the program in BASIC or xBASIC and then translate that working program into the much faster assembly language.

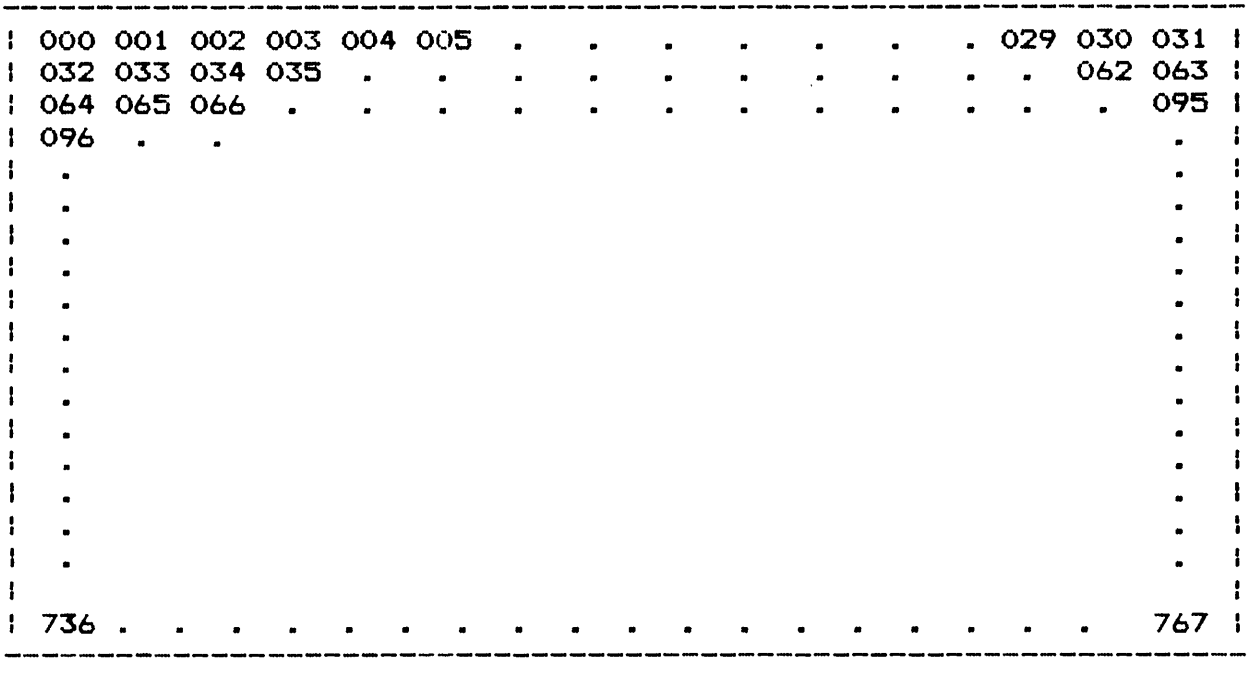
This chapter covers some of the more common BASIC and xBASIC commands, arranged alphabetically. Each command is followed by the source code which duplicates its function. Often, because assembly language is so much freer then BASIC, there will be several ways to accomplish the same task. Of these choices one might be faster, one may take up less memory, and one might be easier to program and understand. When presented with these alternatives, I have selected the example routines which are easiest to program and understand.

CALL CLEAR

The CALL CLEAR BASIC routine clears the screen by placing a space character in all screen positions.

To understand how assembly language accomplishes this we must first understand how the compute creates a 'screen'. The computer has no concept of a screen; it views the screen as one continuous series of memory locations. There are no rows and columns, only 768 possible character locations numbered beginning at the upper left of the screen at 000 and continuing to the bottom right hand corner 767. These memory locations are in VDP RAM beginning at address >0000. Figure 11.0 illustrates this below:

FIGURE 11.0 NUMBERED SCREEN LOCATIONS



To convert a BASIC row and column position into a assembly language graphic screen position we use the following algebraic expression:

$$[\ C + (R*32)\ ] = SP$$

Where 'C' is the column number, 'R' is the row number and 'SP' is the resulting screen position. For example, to find the screen position of (5,7) we simply plug in the values:

$$[\ 5 + (7*32)\ ] = 229$$

Clearing an entire screen is accomplished by placing a space character (32 or >20) in all successive screen locations as demonstrated in the following routine:

```
001 ************************************************************
002 *                                                          *
003 *                      CALL CLEAR                          *
004 * This module will place a space character in all          *
005 * screen positions.                                        *
006 *                                                          *
007 ************************************************************
008         DEF   BEGIN
009         REF   VSBW
010 MYREG   BSS   32          * Reserve memory for my workspace.
011 *
012 BEGIN   LWPI  MYREG       * Set pointer to workspace area.
013         LI    R0,0        * First screen position to print to.
014         LI    R1,>2000    * Load space character.
015         LI    R2,767      * Load our count register.
016 LOOP    BLWP  @VSBW       * Place character on screen.
017         INC   R0          * Increment screen position by 1.
018         DEC   R2          * Decrement our count register.
019         JGT   LOOP        * See if whole screen filled.
020 *
021         END   BEGIN       * End program.
```

Lines 008-012 reserve memory for the Workspace Registers, set the workspace pointer at the beginning of this work area and reference all needed utility programs. Line 013 is the beginning of the working part of the program. It loads R0 with the first screen position to receive a blank character (position 000). Line 014 loads character 32 (the blank space character) into the left byte of R1 as this is the byte that VSBW will utilize. Line 015 sets up R2 as a count register that will reach 0 when all screen positions are filled. Line 016 places the character on the screen and is the beginning of our loop.

The first time this program runs through the 'LOOP' a blank space character will be written to VDP RAM address >0000. Lines 017 and 018 will increase R0 by one and decrease the count register by one. The program will then jump back and write a space character in the next screen location. This will continue until the count register has been decremented to zero. When this happens the program will end. The loop in this program will execute a total of 768 times; filling VDP RAM memory locations 000 through 767 with the value for the space character.

CALL SCREEN

The source code used to color the screen in BASIC is the 'CALL SCREEN' statement. It is quite similar to the source code

we used to mimic the CALL CLEAR routine. The difference is that the foreground and background color of the space character has to be redefined before we fill the screen with it. For example, if we make the foreground and background color of the space character black, then fill the screen with it, it will leave the screen appear black.

The foreground and background color of a character is altered by changing the values of addresses in the Color Table. The Color Table begins at VDP RAM address >0380 and extends to address >039F. Each byte in the Color Table codes for the foreground and background of a group of eight characters. For example, VDP address >0380 holds the byte that codes for the foreground and background colors of character codes 0 through 7. Address >0381 holds the byte that codes for characters 8 through 15. Address >0382 holds the byte that codes for characters 16 through 23. This continues on until address >039F is reached which holds the byte that codes for the final character codes 248 through 255.

Table 11.1 lists the Color Table addresses and character codes each byte holds the color of.

TABLE 11.1 COLOR TABLE ADDRESSES

Table Address	Char. Codes	Table Address	Char. Codes	Table Address	Char. Codes	Table Address	Char. Codes
>0380	0-7	>0384	32-39	>0388	64-71	>038C	96-103
>0381	8-15	>0385	40-47	>0389	72-79	>038D	104-111
>0382	16-23	>0386	48-55	>0390	80-87	>038E	112-119
>0383	24-31	>0387	56-63	>0391	88-95	>038F	120-127
>0390	128-135	>0394	160-167	>0398	192-199	>039C	224-231
>0391	136-143	>0395	168-175	>0399	200-207	>039D	232-239
>0392	144-151	>0396	176-183	>0400	208-215	>039E	240-247
>0393	152-159	>0397	184-191	>0401	216-223	>039F	248-255

The space character is character 32 (HEX >20). Looking at the Color Table outlined in Table 11.1 we see that address >0384 holds the byte that contains the color code for character 32. As we already know there are eight bits in a byte. In the case of a color byte the left most four bits (4 most significant bits) code for the foreground color, while the right four bits (4 least significant bits) code for the background color. From this information we know that if we place a value of >F1 at address >0386 it will set characters 48 through 55 white on black.

The following source code can be used to load a value into a color table address. In this case characters 32 through 39 are set black on black.

```
001  ************************************************************
003  * CALL SCREEN(2)                                           *
004  * PROGRAM MODULE TO LOAD VALUE (BYTE) INTO THE COLOR       *
005  * TABLE, THEREBY SETTING THE FOREGROUND AND BACKGROUND     *
006  * COLOR OF A DESIGNATED CHARACTER SET.                     *
008  ************************************************************
009          REF   VSBW
010  MYREG   BSS   32
011  COLTAB  EQU   >0384
012  COLOR   DATA  >1100
016  BEGIN   LWPI  MYREG
017          LI    R0,COLTAB
018          MOV   @COLOR,R1
019          BLWP  @VSBW
020  *
021          LI    R0,0
022          LI    R1,>2000
023          LI    R2,767
024  LOOP    BLWP  @VSBW
025          INC   R0
026          DEC   R2
027          JGT   LOOP
```

Line 010 sets up the Workspace Register area. Line 011 sets COLTAB equal to >0384, the address in the table we want to write to. Line 012 defines the byte we will use, in this case >11, or black on black. Line 016 starts the program proper. Here we load the address of the Color Table into R0. Line 018 moves the byte we are going to write (>11) into the most significant byte of R1. Line 019 calls the utility program that executes the write. At this point address >0384 now contains the byte >11. Characters 32-39 are now set to black on black.

Lines 021 through 027 are just the CLEAR SCREEN program that prints the space character in all screen positions, but now that character is set to black on black. The screen is now totally black except for the upper and lower border which can be changed by writing a value to VDP Register 7.

DISPLAY AT

To display a message somewhere on the screen in xBASIC you use the simple command:

```
100 DISPLAY AT(4,5):"HIGH"
```

which will put "HIGH" on the screen with the first letter beginning in column 4 row 5 of the screen. As already mentioned, the computer regards the screen as a series of memory locations

numbered 000 to 767. To convert a row and column location into its memory location equivalent use the algebraic expression:

[C + (R*32)] = P

where C is the column, R is the row, and P is the assembly language memory location. Thus location (4,5) becomes:

[4+(5*32)]=164

Now that we know the location on the screen where we want to put the message, we need to know how to store the message in the program until we print it out. This is done through the use of a "TEXT" directive. The following source code outlines the procedure to print something on the screen:

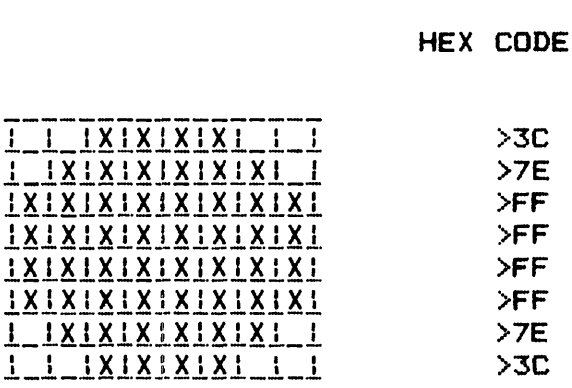
```
001  ************************************************
003  *     DISPLAY AT(6,3):"HOW ARE YOU?"           *
004  *  PROGRAM MODULE TO PRINT A STATEMENT IN A     *
005  *  DESIGNATED SCREEN POSITION.                  *
007  ************************************************
008         REF  VMBW
009  MYREG  BSS  >20
010  ADDR1  TEXT 'HOW ARE YOU?'  * Message to print.
011  *
012  BEGIN  LWPI MYREG
013         LI   R0,102    * [6+(3*32)] Screen location.
014         LI   R1,ADDR1  * Load message.
015         LI   R2,12     * # of characters to write.
016         BLWP @VMBW
017         JMP  $         * Hold display on screen.
```

CALL CHAR

This BASIC statement redefines a specified character using a 16 character HEXadecimal coded string. For example character 33 [>21] is the ASCII value for the exclamation point (!). If we enter the statement:

100 CALL CHAR(33,"FFFFFFFFFFFFFFFF")

then character 33 is redefined as solid square (all areas shaded). If we wanted to redefine a character into a ball shape, we could use the procedure on the following page which outlines a grid to help us create our pattern.



From the figure above it can be seen that the pattern identifier for the 'BALL' is "3C7EFFFFFFFF7E3C". We now construct the following statement:

```
100 CALL CHAR (128,"3C7EFFFFFFFF7E3C")
```

Which defines character 128 as our "ball". We can then place the ball anywhere on the screen with a CALL HCHAR statement. The complete code is thus:

```
100 CALL CHAR(128,"3C7EFFFFFFFF7E3C")
110 CALL HCHAR(4,10,128,1)
```

To understand how assembly language accomplishes the same task we must know where the computer stores patterns. It holds them in a **Pattern Descriptor Table.** This table begins at address >0800 and extends through to address >0FFF in VDP RAM.

Each pattern requires eight bytes to define one character. The pattern of character 0 occupies addresses >0800 through >0807, character 1 occupies addresses >0808 through >080F, character 3 occupies addresses >0810 through >0817 and this continues until the last character, character 256, is reached which occupies addresses >0FF8 through >0FFF.

To quickly find which address begins the code for which character, you can use the following formula:

```
[ 2048 + ( C*8 ) ] = A
```

Where 'C' is the decimal value of the character and 'A' is the decimal value of the desired address. Using this formula we can find that the address that begins the description of character 128 [>80] is :

```
[ 2048 + ( 128 * 8 ) ] = 3072
```

which is VDP address >0C00.

Now that we know the pattern identifier for a ball and the address of where that pattern belongs for character 128, we can write a translation of the following BASIC code:

```
001  ************************************************************
002  *                                                          *
003  *       100 CALL CHAR(128,"3C7EFFFFFFFF7E3C")              *
004  *       110 CALL HCHAR(4,10,128,1)                         *
005  *                                                          *
006  ************************************************************
007         REF  VMBW,VSBW
008  MYREG  BSS  32                        *
009  PATTAB EQU  >0C00                     * [2048+(C*8(()=0C00
010  PAT    DATA >3C7E,>FFFF,>FFFF,>3C7E   * "BALL" pattern
011  *
012  START LWPI MYREG       * Loads the pattern for the balll
013        LI   R0,PATTAB   * into the Pattern Descriptor
014        LI   R1,PAT      * Table.
015        LI   R2,8        *   .
016        BLWP @VMBW       *   .
017  *
018        LI   R0,138      * Places the "ball" (character 128)
019        LI   R1,>8000    * on the screen in position (4,10)
020        BLWP @VSBW       *
021        JMP  $           * Hold display on screen.
```

By adding a few additional lines of code we can repeat the pattern any number of times in the horizontal direction. The following additional lines of source code when placed in the program above will simulate the BASIC statement:

```
CALL HCHAR(4,10,128,8)
```

Replace lines 018 through 025 with the following code:

```
 .
 .         .     .
018        LI    R0,138
019        LI    R1,>8000
020        LI    R2,8      * Count register: loop 8 times.
021  LOOP  BLWP  @VSBW     * Put character on screen.
022        INC   R0        * Next position to place character.
023        DEC   R2        * Decrease count register.
024        JGT   LOOP      * Check if all 8 characters are on
025  *                       screen, if not loop again.
```

To translate the VCHAR statement requires only a slight modification of the code for the HCHAR statement as illustrated on the next page (note only line 022 was altered):

```
 .          .     .
018         LI    R0,138
019         LI    R1,>8000
020         LI    R2,8
021  LOOP   BLWP  @VSBW
022         AI    R0,32      * Increment to screen position
023  DEC    R2               * below last one written to.
024         JGT   LOOP
025  *
```

You will notice that line 022 adds 32 to the current screen position that you are writing to. In this way the next screen location specified is the one directly under the previous one.

This source code, when added to the program lines previously mentioned, is a direct translation of the BASIC statement:

CALL VCHAR(4,10,128,8)

In fact, by altering the amount that you increase *or decrease R0 in your program you can make the patterns print up, down, diagonally or virtually any way by altering this one line of source code.

CALL KEY

This BASIC command sets the keyboard to be tested and returns two variables based on input from the keyboard. The first variable tells you whether or not a key has been pressed, while the second variable returned gives you the value of the key pressed. There is a utility program in assembly language that you can use to return keyboard input. This utility is referred to as KSCAN.

In order to use the KSCAN utility, you have to first determine where you want the input to come from. You can input from the whole keyboard, right side of the keyboard, left side of the keyboard or input from the joysticks.

Address >8374 contains the byte that determines which keyboard device you want to select. The following values select for desired keyboard devices:

>00	Checks the entire keyboard.
>01	Checks left side of keyboard and joystick #1.
>02	Checks right side of keyboard and joystick #2.

From the above table we see that if a value of >01 is placed at address >8374 the KSCAN routine will check for input from the left side of the keyboard as well as input from joystick #1.

When a key is pressed its ASCII value is placed at address >8375. If no key was pressed this address will contain >FF. Lets

consider a program where input from the keyboard is used to perform some task. The following BASIC program will print a message on the screen based on which arrow key has been pressed.

```
100 CALL KEY (1,KEY,STATUS)
110 IF STATUS=0 THEN 100
120 IF KEY=5 THEN A$="UP KEY PRESSED"
140 IF KEY=3 THEN A$="RIGHT KEY PRESSED"
160 IF KEY=0 THEN A$="DOWN KEY PRESSED"
180 IF KEY=2 THEN A$="LEFT KEY PRESSED"
190 DISPLAY AT (4,10):A$
200 GOTO 100
```

This program will display the "UP KEY PRESSED" message if the up 'E' key is pressed. If the 'D' key is pressed the "RIGHT KEY PRESSED' message appears. This continues on for the other two keys (X & S). The assembly language translation of this program illustrating the CALL KEY function is as follows:

```
001 ***************************************************************
002 *                    CALL KEY (1,KEY,STATUS)                  *
003 * This module will input from the arrow keys (E,D,X,S)        *
004 * and display a message indicating the pressed key.           *
005 ***************************************************************
008         DEF   BEGIN            * Reference needed utilities.
009         REF   KSCAN,VMBW       * Address to select keyboard
010 KBOARD EQU   >8374             * Holds ASCII # of pressed key
011 KEY     EQU   >8375            *
012 *
013 KEYUP   BYTE  5                *            ASCII values
014 KEYRT   BYTE  3                *          of E, D, X and S
015 KEYDN   BYTE  0                *              keys
016 KEYLT   BYTE  2                *
017 HEXFF   BYTE  >FF              * No key pressed value
018 ONE     BYTE  1                *
019 *
020 UP      TEXT  'UP KEY PRESSED   '
021 RIGHT   TEXT  'RIGHT KEY PRESSED'
022 DOWN    TEXT  'DOWN KEY PRESSED '
023 LEFT    TEXT  'LEFT KEY PRESSED '
024         EVEN
025 *
026 MYREG   BSS   >20
027 *
028 BEGIN   LWPI  MYREG
029         MOVB  @ONE,@KEYBOARD * Check left side of keyboard.
030 LOOP    BLWP  @KSCAN           * Check for keyboard input.
031         CB    @HEXFF,@KEY      * Was a key pressed?
032         JEQ   LOOP             *
033         CB    @KEYUP,@KEY      * Compare to see which
034         JEQ   PUP              *   arrow key was pressed.
```

```
035          CB    @KEYRT,@KEY    *
036          JEQ   PRIGHT         *
037          CB    @KEYDN,@KEY    *
038          JEQ   PDOWN          *
039          CB    @KEYLT,@KEY    *
040          JEQ   PLEFT          *
041          B     @LOOP          *  If key not found, LOOP again.
042 PUP      LI    R1,UP          *  Load
043          B     @PRINT         *      correct
044 PRIGHT   LI    R1,RIGHT       *              message
045          B     @PRINT         *                      into
046 PDOWN    LI    R1,DOWN        *                          R1
047          B     @PRINT         *
048 PLEFT    LI    R1,LEFT        *
049          B     @PRINT         *
050 *
051 PRINT    LI    R0,138         * Print
052          LI    R2,17          *     message on
053          BLWP  @VMBW          *                screen
054          B     @LOOP          * Repeat program
                                  * End program.
```

CALL JOYST

If you place a value of >01 at address >8374 the KSCAN routine will check for input from joystick #1 (as well as from the left side of the keyboard). If you place a value of >02 at address >8374 the KSCAN utility will check for input from joystick #2 (as well as from the right side of the keyboard). Input from joysticks is placed into CPU addresses >8376 (Y-position) and >8377 (X-position). Table 11.2 lists the possible values that may be returned.

TABLE 11.2 JOYSTICK INPUT Y POSITION

Joystick Y Position	Value Returned	Address
CENTER	>00	>8376
UP	>04	>8376
DOWN	>FC	>8376

TABLE 11.3 JOYSTICK INPUT X POSITION

Joystick X Position	Value Returned	Address
CENTER	>00	>8377
RIGHT	>04	>8377
LEFT	>FC	>8377

Lets assume that a value of >01 is at address >8374. Lets also assume that joystick #1 is in the DOWN-RIGHT position. When the KSCAN routine is called a value of -4 (>FC) is placed at address >8376 and a value of 4 (>04) is placed at address >8377.

The following xBASIC program will print out a message on the screen reporting on the current position of joystick #1. It is very similar to the CALL KEY program that was presented earlier.

```
100 CALL JOYST(1,JOYX,JOYY)
110 IF JOYY=0 AND JOYX=0 THEN A$="CENTER"
120 IF JOYY=4 AND JOYX=0 THEN A$="UP"
130 IF JOYY=4 AND JOYX=4 THEN A$="UP-RIGHT"
140 IF JOYY=0 AND JOYX=4 THEN A$="RIGHT"
150 IF JOYY=-4 AND JOYX=4 THEN A$="DOWN-RIGHT"
160 IF JOYY=-4 AND JOYX=0 THEN A$="DOWN"
170 IF JOYY=-4 AND JOYX=-4 THEN A$="DOWN-LEFT"
180 IF JOYY=0 AND JOYX=-4 THEN A$="LEFT"
190 IF JOYY=4 AND JOYX=-4 THEN A$="UP-LEFT"
200 DISPLAY AT(4,10):A$
210 GOTO 100
```

The above program will display a message on the screen reporting on the current position of joystick #1. The source code that follows is a direct translation of the previous xBASIC program. You may wish to study it in great detail as most game programs utilize a joystick input of one type or another.

```
001 ************************************************************
003 *                CALL JOYST(1,JOYx,JOYY)                   *
004 * This module will input from joystick #1 and display its *
005 * current position on the screen.                          *
007 ************************************************************
008         DEF   START
009         REF   KSCAN,VMBW
010 *
```

```
011  KBOARD  EQU   >8374  * Address of keyboard device select.
012  JOYY    EQU   >8376  * Joystick input "Y" value.
013  *
014  JOYUP   BYTE  4,0        *
015  JOYUR   BYTE  4,4        *
016  JOYRT   BYTE  0,4        *
017  JOYDR   BYTE  -4,4       *
018  JOYDN   BYTE  -4,0       *
019  JOYDL   BYTE  -4,-4      *
020  JOYLT   BYTE  0,-4       *
021  JOYUL   BYTE  4,-4       *
022  JOYCT   BYTE  0,0        *
023  HEXFF   BYTE  >FF        *
024  ONE     BYTE  1          *
025  *
026  UP      TEXT  'UP             '      *
027  UPRT    TEXT  'UP-RT          '      *   JOYSTICK
028  RT      TEXT  'RIGHT          '      *
029  DNRT    TEXT  'DOWN-RIGHT     '      *       POSITION
030  DN      TEXT  'DOWN           '      *
031  DNLT    TEXT  'DOWN-LEFT      '      *           MESSAGES
032  LT      TEXT  'LEFT           '      *
033  UPLT    TEXT  'UP-LEFT        '      *
034  CENTER  TEXT  'CENTER         '
035  EVEN
036  *
037  MYREG   BSS   32      * Reserve space for Workspace
038  *                       Registers.
039  BEGIN   LWPI  MYREG                 * Load pointer.
040          MOVB  @ONE,@KBOARD          * Select keyboard device.
041  *
042  START   BLWP  @KSCAN                * Scan joystick..
043          C     @JOYY,@JOYUP
044          JEQ   P1
045          C     @JOYY,@JOYUR
046          JEQ   P2                    * Compare to see what
047          C     @JOYY,@JOYRT
048          JEQ   P3                    * the X and Y position
049          C     @JOYY,@JOYDR
050          JEQ   P4                    * of the joystick is.
051          C     @JOYY,@JOYDN          *
052          JEQ   P5                    *
053          C     @JOYY,@JOYDL          *
054          JEQ   P6                    *
055          C     @JOYY,@JOYLT          *
056          JEQ   P7                    *
057          C     @JOYY,@JOYUL          *
058          JEQ   P8                    *
059  *
060          LI    R1,CENTER             *
061          B     @PRINT                *
```

```
062  P1      LI    R1,UP              *
063          B     @PRINT             *
064  P2      LI    R1,UPRT            *  Load
065          B     @PRINT             *
066  P3      LI    R1,RT              *       appropriate
067          B     @PRINT             *
068  P4      LI    R1,DNRT            *                 message
069          B     @PRINT             *
070  P5      LI    R1,DN              *
071          B     @PRINT             *
072  P6      LI    R1,DNLT            *
073          B     @PRINT             *
074  P7      LI    R1,LT              *
075          B     @PRINT             *
076  P8      LI    R1,UPLT            *
077  *
078  PRINT   LI    R0,138             * Display
079          LI    R2,10              *    message on
080          BLWP  @VMBW              *          screen.
081          B     @START             * Return and check again.
082          END   BEGIN
```

DIM

BASIC is a powerful language when it comes to automatic string manipulation, array handling and specific error messages letting you know exactly where you went wrong. The price you pay for these luxuries is that the BASIC program will run very slowly when compared with assembly language. Because array management is not directly handled by the computer when using assembly language, you will have to set memory aside for that purpose. The best way to do this is through the use of the BSS and BES directives, either of these directives will set aside any amount of memory. Handling these 'chunks' is not too difficult, but it may help to use a pen and paper to keep track of your own arrays as you set them up in memory.

FOR—NEXT

The FOR-NEXT statement in BASIC can be used to create a delay loop or a counting loop. For example, if you want to put something on the screen for someone to read you might incorporate a "delay loop" to hold the message on the screen for a period of time.

In game programming with assembly language these delays become much more important because the program executes so quickly that an object on the screen could move so quickly that it would de visible only as a blur. The source code on the next page outlines a simple delay loop.

```
001  ********************************************************
003  *      FOR DELAY=1 TO 1000 :: NEXT DELAY               *
005  ********************************************************
006          .
007          .
008          LI    R1,1        * For 1
009          LI    R2,1000     * To 1000
010 DELAY    DEC   R2
011           C     R1,R2
012          JNE   DELAY       * Next Delay
013          .
```

Of course this delay loop will execute much more quickly then its BASIC counterpart. In fact, unless you were looking for it you would probably not even notice this small of a delay!

The maximum value we can use in a single delay loop like the one in the previous example is 32767. To loop with larger numbers we can create two registers working together to keep count. In the next example, the first register counts down from 32767 and then R2 clicks in to repeat the count for a total delay of 98301 "loops".

```
001  ********************************************************
003  *        FOR DELAY-1 TO 98301 :: NEXT DELAY            *
005  ********************************************************
006          .
007          .
008          LI  R2,3
009  LOOP1   LI  R1,32767  * Load a count register.
010  LOOP2   DEC R1        * Load maximum delay.
011          JNE LOOP2
012          DEC R2
013          JEQ OUT
014          JMP LOOP1
015  OUT     .
```

Here we use R2 as our "second count" register and we use R1 as our "primary count" register. Line 009 is the beginning of our loop, R1 is loaded with the maximum signed value it can hold. The next line (010) decrements R1 by one and line 011 tests to see if R1 is zero yet. If not, the program jumps to LOOP2 and decrements R1 again. This continues until R1 is equal to zero, then R2 is decremented. If R2 has been decremented to zero program control jumps to OUT, otherwise the program jumps to LOOP1 and R1 is reloaded and the delay continues.

FOR-NEXT-STEP

For this instruction you just increment your counter register the amount of the step as demonstrated in the following source code:

```
001  ************************************************************
003  *        FOR DELAY=0 TO 75 STEP 3 :: NEXT DELAY            *
005  ************************************************************
006           .
007           .
008          LI    R1,0
009  DELAY   INCT  R1
010          INC   R1
011          CI    R1,75
012          JNE   DELAY
```

Notice that lines 009 and 010 of the last example increment our count register (R1) a total of three for each pass of the DELAY loop. Take note that this source code could also be written:

```
.            .   .
008          LI  R1,0
009  DELAY   AI  R1,3
010          CI  R1,75
011          JNE DELAY
```

Either version would work equally as well.

For very large numbers we can again use two counter registers to keep track of things. Following our first example above we could translate the xBASIC statement FOR I=10000 TO 0 STEP -1 into the source code:

```
001  ************************************************************
003  *     FOR I=10000 TO 0 STEP -2 :: NEXT I                   *
005  ************************************************************
006           .
007           .
008          LI    R2,10
009  LOOP1   LI    R1,1000
010  LOOP2   DECT  R1
011          JNE   LOOP2
012          DEC   R2
013          JNE   LOOP1
```

Here we see R1 decremented by two after each loop. If you were using the value of "I" for some other procedure in the program you could get it simply by multiplying R1 and R2 together at any point during execution of these loops.

IF—THEN—ELSE

Conditional jumps and compare instructions constitute the primary computing structure in assembly language. It is fairly straight forward and can be easily demonstrated with a translation of the following:

```
001  ************************************************************
003  *            IF DAMAGE=100 THEN SHIP=10                    *
005  ************************************************************
 .
008  DAMAGE DATA >0000
009  SHIP   DATA >0000
 .   .      .
200  SUB1   MOV  @DAMAGE,R1   *
201         CI   R1,100        * If DAMAGE=100
202         JNE  OUT1          * Then...
203         LI   R1,10         *
204         MOV  R1,@SHIP      *          ..SHIP=10
205  OUT1   RT
```

To add an ELSE to the statement you simply add three additional lines of source code as follows:

```
001  ************************************************************
003  *         IF DAMAGE=100 THEN SHIP=10 ELSE SHIP=5           *
005  ************************************************************
 .
008  DAMAGE  DATA  >0000
009  SHIP    DATA  >0000
 .           .     .
200          MOV   @DAMAGE,R1
201          CI    R1,>64
202          JNE   ELSE1
203          LI    R1,>A
204          MOV   R1,@SHIP
205          JMP   OUT1
206  ELSE1   LI    R1,>5
207          MOV   R1,@SHIP
208  OUT1    RT
```

ON GOSUB

In BASIC, you are limited with the GOSUB instruction to test very specific values before proceeding. For example:

```
100 ON Y GOSUB 200,230,240
```

In this example Y must be 1 or 2 or 3. Only one branch test is

performed with control returning to the statement just after the GOSUB after that one branch is finished. Also, if Y was not equal to any of the branches (ie: not=1, 2, or 3) an error message would be returned by the computer.

Assembly language permits you much greater freedom in programming in that it permits multiply branch testing. In this situation, one, two or all the branches might be executed. Or alternatively, none of the branches may be branched to under certain conditions. The source code on the following page could be found in a game program where some value, perhaps inputed from the keyboard, determines which subprogram is branched to.

```
001 ************************************************************
003 *              ON VALUE GOSUB 100,200,300                  *
004 * Program module to perform a multiple branch test         *
006 ************************************************************
007          .
008          .
009          MOV   @VALUE,R0
010          CI    R0,100          * See if VALUE=100
011          JNE   NEXT1           * If not, then jump to NEXT1
012          BL    MISS            * Branch & Link w/ MISS routine
013 NEXT1    CI    R0,200          * See if VALUE=200
014          JNE   NEXT2           * If not, then jump to NEXT2
015          BL    HIT             * Branch & Link w/ HIT routine
016 NEXT2    CI    R0,300          * See if VALUE=300
017          JNE   OUT             * If not, jump to OUT
018          BL    KILLED          * Branch & Link with KILLED
```

You will be BLing out of the program and RTing back to within the multiple branch test above to continue until all the branches have been tested. You will have to be careful that your subprograms MISS, HIT and KILLED do not change the value in R0 or an accidental triggering of another branch may occur.

ON GOTO

This is another version of the GOSUB structure we have just covered. The difference is that after one branch meets with a successful test, control jumps back to the point following all the branch tests.

```
001 ************************************************************
002 *                       ON GOTO                              *
003 * This program module allows you to test branches one at    *
004 * a time. Program control transfers to a point following    *
005 * all branch tests after completion of a subroutine.       *
007 ************************************************************
010          .
011          .
```

```
012           MOV   @VALUE,R0
013           CI    R0,100
014           JNE   NEXT1
015           JMP   SUBR1          *
016 NEXT1     CI    R0,200
017           JNE   NEXT2
018           JMP   SUBR2          *
019 NEXT2     CI    R0,300
020           JNE   OUT            *
021           JMP   SUBR3          *
022 OUT    [all subroutines "JMP" to location OUT when finished]
```

Instead of RT, each subroutine in the last example will JMP back to location OUT, which lets the program continue without running through any more tests of the branches. In this way no branch is accidentally triggered if the subroutine were to change the contents of R0.

REM

You can make notes directly inside program by preceding them with an asterisk (*). An entire line in a source program may be reserved in this way for comments or notes about your program. Comments also can be made after the operand field in most instructions by spacing once and typing in an asterisk (*) followed by your note or comment. The asterisk serves as a signal to the assembler to ignore the information you have typed. Your remarks remain part of the source code only and are omitted during the assembly process.

RETURN

There are two return instructions in assembly language. They operate very similar to the way RETURN does in BASIC. THE RTWP takes you back from a subprogram to just after the BL (GOSUB) instruction that sent you to a subroutine.

When a BL or BLWP instruction is reached, the address which immediately follows the BL or BLWP instruction itself is placed in R11. That address then stays in R11 until a RT or RTWP is encountered. When this occurs, the address is taken from R11 and placed into the Program Counter. This transfers program control back to the instruction just after the BL or BLWP line.

RUN

If you are not going from BASIC to an assembly program, but are only running an assembly program by itself, there are basically two ways to run the program using the Editor/Assembler. The first way is to define an entry point with the DEF statement at the

beginning of the program. Using this method you load the object code into the computer using the LOAD and RUN option of the Editor/Assembler module. After the program is loaded you press ENTER and the PROGRAM NAME? prompt appears. You then type in the starting point of program. This entry must match a entry in the DEF statement at the beginning of the program.

The second way to run a assembly language program is to place the entry point of the program in the operand field of an END directive. When this program is loaded it will start running automatically as soon as the file is loaded. The following illustrates these two methods of starting assembly programs:

```
001          DEF START
 .           .     .
 .           .     .
020  START   .     .
```

Using this procedure you must load the file that contains the object code with the LOAD and RUN option of the Editor/Assembler. When the file is loaded hit ENTER and the PROGRAM NAME? prompt appears. You then type in the entry point in your program which also must be found in a DEF statement at the beginning of the program.

```
020  START   .     .
 .           .     .
 .           .     .
800          END   START
```

Placing the entry point to your program in the operand field of a END statement causes the program to start running automatically as soon as it is loaded with the LOAD and RUN option of the Editor/Assembler.

12

LINKING WITH BASIC

Many times in programming you will want to add an assembly language module to a BASIC program. This has the effect of allowing you to create your "own" BASIC commands which you can use as needed. You can also add fast-executing modules at specific points to speed up program execution. This chapter will discuss in detail the ways in which you can link your BASIC programs with assembly language programs.

Both the Editor/Assembler module and the Mini Memory module provide you with several additional BASIC commands. These commands are designed to aid you in the task of interfacing your assembly language programs with BASIC. Table 12.0 outlines these commands.

TABLE 12.0 BASIC ASSEMBLY LANGUAGE SUPPORT COMMANDS

Command	Description
CALL INIT	Initializes CPU memory for AL subroutines
CALL LOAD	Load data or AL program into CPU RAM memory.
CALL LINK	Link BASIC program with AL program.
CALL PEEK	Look at data in a CPU RAM address.
CALL PEEKV	Look at data in a VDP RAM address.
CALL POKEV	Load data into VDP RAM.
CALL CHARPAT	Return the value of a character pattern.

Each preceding BASIC command is discussed in detail in the sections that follow in this chapter.

CALL INIT

This command must be called before any assembly language programs are loaded through the BASIC program. This command should not be called once the assembly language program is loaded or the program will be rendered inaccessible. The CALL INIT command goes through the following procedures when called:

1. Check to see if memory expansion is connected to the console.

2. Loads utility routines from the Editor/Assembler module into the memory expansion starting at address >2000.

3. Loads the REF/DEF tables into the memory expansion at addresses >3F38 through >3FFF.

If you use the command CALL INIT with the mini memory module, all programs and data are erased. CALL INIT also initializes CPU RAM for assembly language subroutines and re-initializes the internal tables of the mini memory module. If memory expansion is attached, access is enabled to both the module and memory expansion. If the memory expansion is not connected or turned off, the memory expansion is not recognized. You do not need to use CALL INIT each time you use the module, since it has its own internal power supply. **Remember that all data and programs on the module are lost when you use the CALL INIT command!**

CALL LOAD

There are two ways in which the CALL LOAD command can be used. The first is to load an assembly language object code file, and the second is to load or "poke" data directly into CPU RAM.

LOADING OBJECT CODE

To load an assembly language program (object code) you would use the following format of the CALL LOAD statement:

```
CALL LOAD("device.filename")
```

where the device.filename is a string expression such as DSK1.FILE1. This file must be object code. You can load more than one object file at a time by separating the files you want by commas as in the following example:

```
CALL LOAD("DSK1.FILE1","DSK1.FILE2")
```

which loads the two files FILE1 and FILE2 from disk drive 1.

Relocatable object code is loaded at the first available address. With no files loaded and memory expansion attached this address is >A000. When using the mini memory module without the memory expansion unit attached this address is >7118, the lowest available address in the module's RAM. Subsequent programs are loaded in a sequential manner, with the next program loaded in memory immediately following the previous program. Absolute code is loaded at the absolute address specified by the object code. Your program should not use absolute code unless extreme care is taken, as loading data into an area of memory used by the TI BASIC interpreter can cause the computer to "crash".

"POKING" DATA

To load or "poke" data into an area of CPU RAM you would use the following format of the CAll LOAD command:

```
CALL LOAD(address,value)
```

where the address is a decimal number which can be any value from -32768 through 32767. Values 0 through 32767 represent addresses 0000 through 7FFF, while the values -32768 through -1 represent 8000 through FFFF expressed as two's compliment form. In order to find an address above 32767 you must subtract 65536 from it. You can load any number of bytes beginning at an address by specifying the values to load. For example, the statement:

```
CALL LOAD(-36864,24,13,90)
```

loads the values >18, >0D and >5A into the respective bytes at locations >7000, >7001 and >7002.

You can specify more than one poke list by separating the last byte of one poke list and the starting address of the next poke list with a pair of quotes as in the next example

```
CALL LOAD(-36864,24,13,"",53248,19)
```

which loads the same values as the preceding example and also loads the value >13 into address >D000.

You could also load an assembly language program byte-by-byte in this manner by poking in the various instructions. However to run a machine language program loaded in this manner you would have to enter the program name and starting point into the REF/DEF table so that the computer could find it. You do not need to worry about these steps if your program was loaded by the Editor/Assembler loader since that is done for you. If you are using the Mini Memory Module you should use the procedure outlined on page 144. If you use xBASIC to run your assembly language program you must first perform the following steps:

1. Read the First Free Address in the Module with the CALL PEEK command. The FFAM can be found at address >2028.

2. Read the Last Free Address in the Module. This address can be found at address >202A.

3. Subtract the FFAM from the LFAM. If they differ by at least 8 bytes, there is room to add your program name and address.

4. Use the CALL LOAD command to change the LFAM to a value 8 bytes less then its old value.

5. Use the CALL LOAD command to load the program name (6 bytes in length) starting at the new LFAM followed by two bytes which give the program starting address.

For example, suppose the LFAM is >8000, your program name is FILE. The program begins at addess >8300. You would then load the following information:

```
CALL LOAD(28700,127,251)
CALL LOAD(32763,70,73,76,69,32,32,131,00)
                              / /
                  NAME PADDED TO 6 CHARACTERS
```

CALL LINK

The CALL LINK command lets you pass control from a BASIC program to an already loaded assembly language program. It also lets you optionally pass a list of parameters from the BASIC program to the assembly language program.

The format for the CALL LINK command is as follows:

```
CALL LINK("program-name","parameters...")
```

The **program-name** is a 1 to 6 character string that defines the entry point into the program. It must appear in the REF/DEF Table of the assembly language program that you are trying to link with. The assembly language program must already be in memory (loaded via the CALL LOAD command).

The **parameters** are optional. They allow you to pass string variables, numeric variables, or expressions between your BASIC and assembly language programs. For example, the statement:

```
CALL LINK("BEGIN",A,D$)
```

passes control from a BASIC program to the assembly language program BEGIN, with the numeric variable 'A' and the string variable 'D$' passed to it.

The CALL LINK command goes through the following operations when called:

1. Check to see if AL program name is 1 to 6 characters in length.

2. If name is right length, the name is looked up in the REF/DEF Table, beginning at the lowest address. The program name is then pushed onto the value stack.

   note: An error is generated if there are duplicate names in DEF instructions.

3. If parameters are to be passed the utility will build an argument list. This list identifies the type of arguments and builds a stack entry for each argument.

4. Program control is transferred to the assembly language program through a direct AL "branch" instruction.

note: In order to return to your BASIC program, your AL program must preserve and restore the values in Workspace Registers R11, R13, R14, and R15 before ending.

5. At the end of the assembly language program, control will return to the calling BASIC program unless an error has occurred. If an error has occurred, the program branches to an error routine.

note: Address >8310 contains the value stack pointer in use by BASIC interpreter.

PARAMETER PASSING WITH CALL LINK

Up to 16 arguments can be passed between a BASIC program and an assembly language program. If the parameter is an expression, it is passed by its value, if it is a variable it is passed by name. Any variable except an expression can have its value changed by the assembly language program. This value, in turn, can be passed back to the BASIC program.

You can pass entire arrays by enclosing them in parentheses. Arrays with more than one dimension are indicated by placing commas between the parentheses to indicate the number of dimensions. The following is an example outlining several simple variables (simple variables do not include expressions):

```
CALL LINK("BEGIN",A,B$,SCORE,F$(),G$(,))
        A    =   numeric variable
       B$    =   string variable
    SCORE    =   numeric variable
     F$()    =   one-dimensional array
    G$(,)    =   two-dimensional array
```

If you need to pass variables to your assembly language program but do not need to change their values, surround the variable with parentheses. Arrays however, can not be passed in this manner. For instance, all but the last two in the last example can be passed without having their value changed on return to the calling BASIC program as outlined below:

```
CALL LINK("BEGIN",(A),(B$),(SCORE))
```

Also, constants such as SCORE-3, do not have their values changed by the assembly language program on return to BASIC.

Arguments are passed to an assembly language through an identifier list in CPU RAM. It is not necessary for you to have a knowledge of how arguments are passed if you use the utilities described in section 13.1 If you want to delve deeper and

construct your own utilities, see pages 278-280 of your Editor/Assembler manual.

CALL PEEK

The CALL PEEK command allows you to read bytes of CPU RAM directly into BASIC variables. The following statement is an example of the format of the CALL PEEK command:

```
CALL PEEK(address,variable....)
```

where the address is a decimal number which can be any value from -32768 through 32767. Values 0 through 32767 represent addresses >0000 through >7FFF, while the values -32768 through -1 represent >8000 through >FFFF expressed as two's compliment form. In order to find an address above 32767 you must subtract 65360 from it. You can peek into any number of successive bytes of CPU RAM by simply specifying the variables.

The following example illustrates how data can be read from CPU RAM:

```
CALL PEEK(-36864,A,B,C,D)
```

This statement lets 'A' represent the value held at address >7000, 'B' the value at address >7001, 'C' the value at address >7002 and 'C' the value at address >7003.

You can read from more then one address in a single PEEK statement by separating the last variable of one PEEK list and the Beginning PEEK address of the next list with a pair of quotes. This is illustrated as follows:

```
CALL PEEK(53248,A,B(3),"",-36864,C)
```

This statement lets 'A' and the third element in the array designated 'B' represent the values at addresses >D000 (53248) and >D001 (53248) respectively.

CALL PEEKV

The CALL PEEKV command is used to read bytes of data from VDP RAM. It works in exactly the same manner as the CALL PEEK command except that CALL PEEKV will read from VDP RAM. The format of the CALL PEEKV is the following:

```
CALL PEEKV(address,variable,var...)
```

The address is a **decimal** number which can range in value from 0 through 16383. The values 0 through 16383 represent addresses >0000 through >3FFF in VDP RAM. If you try to access a higher

address then >3FFF the system will crash requiring you to turn the power off and back on again in order to continue.

The following example illustrates the use of the CALL PEEKV command:

```
CALL PEEKV(768,A,B(2),"",10,C)
```

This statement will read a value from VDP RAM address >0300 into 'A' and a value from VDP RAM addres >0301 into the second element of the numeric array designated 'B'. A value will also be read from VDP RAM address >000A into 'C'.

CALL POKEV

The CALL POKEV command allows you to read bytes of VDP RAM directly into BASIC variables. It works in exactly the same manner as the CALL POKE command, except that CALLPOKEV will poke data into VDP RAM instead of CPU RAM. The format of the CALL POKEV command is as follows:

```
CALL POKEV(address,variable...)
```

where the address is a decimal number which can be any value from 0 through 16383. Values 0 through 16383 represent addresses >0000 through >3FFF. Keep in mind that VDP RAM only has 16K of memory. If you try to poke a value into an address higher than >3FFF, the system will crash requiring you to turn the console off and back on in order to continue.

The following example:

```
CALL POKEV(300,32,32,32,"",5,SCORE)
```

places the value 32 (>20) in VDP RAM addresses 300 (>012C), 301 (>012D), and 302 (>012E). It also places the value of SCORE in VDP RAM address 5 (>0005).

CALL CHARPAT

The CALL CHARPAT command returns a 16-character pattern identifier that codes for the character specified by the character-code. The format of the CALL CHARPART command is as follows:

```
CALL CHARPAT(character-code,string-variable)
```

where the character-code is any character number from 32 to 159. The pattern identifier codes for the ASCII character set normally occupy character codes 32 through 95, although you can redefine and can be defined through the use of the CALL CHAR command.

PARAMETER PASSING

Besides the additional BASIC commands provided, the Editor/Assembler and Mini Memory module also provide several assembly language utility programs that greatly simplify passing arguments between AL and BASIC. You can also return errors that occurred during execution of an assembly language module. Table 12.1 outlines these utilities.

TABLE 12.1 BASIC INTERFACE UTILITIES

UTILITY	DESCRIPTION
NUMSAG	Number Assignment.
STRASG	String Assignment.
NUMREF	Number Reference.
STRREF	String Reference.
ERR	Error reporting routine.

If you are using the Editor/Assembler these utility programs can be found on the disk labeled 'A' in the file named BSCSUP. They are in relocatable code and are about 900 bytes long. To use them you must include them in a REF statement at the beginning of your program. In order to load them you must place the statement:

```
CALL LOAD("DSK1.BSCSUP")
```

in your BASIC program.

If you are using the Mini Memory module, the addresses of these utilities can be found on page 148.

RADIX 100 NOTATION

The values of variables passed from BASIC to assembly language programs are stored in the Floating Point Accumulator which begins at VDP RAM Address >834A. Before we progress to the utility programs proper, we must explain radix 100 notation.

In radix 100 notation all numbers range from 1.000000000000 through 99.000000000000 multiplied by 100 raised to a power ranging from -64 to 64.

Each number is coded for in an 8 byte "value stack" located in VDP RAM. The first byte in the value stack indicates the exponent of the numerical value. If the exponent is positive, the byte

value is 64 more than the exponent. If the exponent is negative, the byte value is gotten by subtracting 64 from the exponent. For example, if the exponent is 3, the byte is 67 or >43. If the exponent is -2, the byte is 62 or >3E. If the exponent is negative, the first two bytes are entered in two's-compliment form.

After the exponent byte, the remaining seven bytes in the value stack contain the value of the number. No regard is given to the decimal point when transforming numbers into their hexadecimal equivalents. The second through eighth byte for a radix 100 value of:

$$100^{3} \times 23.456$$

is constructed as follows:

$100^{3} \times 23$	45	60	00	00	00	00
>43 >17	>2D	>3C	>00	>00	>00	>00

The following examples illustrates how several different numbers would be written in radix 100 notation and how the value stack would be structures in each case.

TABLE 13.2 EXAMPLES OF CONVERSION TO RADIX 100 NOTATION

Decimal Value	Radix 100 Notation	Value Stack
6	6×100^{0}	>40 >06 >00 >00 >00 >00 >00 >00
60	60×100^{0}	>40 >3C >00 >00 >00 >00 >00 >00
1,234,560	1.23456×100^{3}	>43 >01 >17 >2D >3C >00 >00 >00
12,345,600	12.3456×100^{3}	>43 >0C >22 >38 >00 >00 >00 >00
0*	0×100^{0}	>00 >00 >XX >XX >XX >XX >XX >XX
-6	-6×100^{0}	>BF >FA >00 >00 >00 >00 >00 >00
-60	-60×100^{0}	>BF >C4 >00 >00 >00 >00 >00 >00
-1,234,560	-1.23456×100^{3}	>BC >FF >17 >2D >3C >00 >00 >00

*Zero is expressed by >00 in the first two bytes & undefined in the remaining 6 bytes.

(NUMASG) NUMBER ASSIGNMENT

This utility allows you to assign a value to a variable passed as an arguement via the CALL LINK command of BASIC.

Follow the steps outlined below in order to use this utility.

1. Place a value of 0 in R0 if the variable is a simple variable. If the variable is an element in an array, place the element number in R0.

   Note: With OPTION BASE 0 (BASIC default) the array elements are numbered starting at 0. If OPTION BASE 1 is selected the array elements are numbered starting at 1.
   Element numbers for multiple dimension arrays are found by counting through the first level, then the second level and so on. For example, an array defined as X(6,6,6) with an OPTION BASE of 0; element number X(3,2,1) is found:

$$(3 * 7^{2}) + (2 * 7^{1}) + (1 * 7^{0}) = 162 = \text{element \#}$$

2. Place the arguement number as a full word in R1. The arguement number is at it appears in the arguement list of the CALL LINK statement.

   Note: The arguement number is the order in which the arguement appears in the parameter list of the CALL LINK statement. For example, in the statement:

```
CALL LINK("BEGIN",X,Y,Z)
```

   'X' is arguement #1, 'Y' is arguement #2, and 'Z' is arguement #3

3. Enter the value you want to assign into the Floating Point Accumulator which begins at address >834A. The number must be in Radix 100 notation.

4. Access the utility by BLWP @NUMASG using the Editor/Assembler or BLWP @6040 if you are using the Mini Memory Module.

For example, the statement CALL LINK("FILE1",X,Y,Z) when encountered in BASIC would pass control to the assembly language program FILE1. If the Floating Point Accumulator beginning at address >834A contains >43 >02 >22 >38 >00 >00 >00 >00, R0 contains >00 and R1 contains >02, then BLWP @NUMASG assigns 2,345,600 to 'Y'.

The following source code can be used to load a value into the FAC area:

```
.
.
FAC     EQU   >834A
VALUE   BYTE  >XX,>XX,>XX,>XX,>XX,>XX,>XX,>XX
.
        LI    R1,FAC
        LI    R2,VALUE
        LI    R3,4
LOOP    MOV   *R2+,*R1+
        DEC   R3
        JNE   LOOP
.
```

(STRASG) STRING ASSIGNMENT

This utility allows you to assign a string to a string variable passed via BASIC command CALL LINK. Before using this utility you must:

1. Create the string in CPU RAM with the first byte in the string indicating the length of the string.

2. For simple string variables, place a value of 0 in R0. If you are assigning a string to an array; place the array element number in R0.

3. Place the address of the string in R2.

4. Place the arguement number as a full word in R1.

5. Access the utility with BLWP @STRASG if using the Editor/Assembler or BLWP @>6048 if you are using the Mini Memory Module.

The example outlined below demonstrates the usage of the STRASG utility. The string "HELLO" is assigned to the string variable A$ which is displayed on return to BASIC.

```
001         DEF  START
002         REF  STRASG
003 MESS    BYTE >05
004         TEXT 'HELLO'
005 START   CLR
006         LI
007         LI
008         BLWP @STRASG
009         RT
010         END
```

The following is the BASIC program that is needed. If you are using the Mini Memory module, omit line 20 as the program is already in memory. You would also need to change line 010 of the source code and omit lines 001 and 002.

```
10  CALL INIT
20  CALL LOAD("DSK1.BSCSUP","DSK1.START")
30  CALL LINK("START",A$)
40  PRINT A$
```

[NUMREF] NUMBER REFERENCE

This utility allows you to get the value of a variable passed into your assembly language program through CALL LINK. In order to do this you need to follow the following steps:

1. If it is a simple variable, place 0 in R0. If it is an array element, place the element number in R0.

2. Place the arguement number as a full word in R1.

3. Call the utility via BLWP @NUMREF or BLWP @>6044.

The value of the variable will be returned in the Floating Point Accumulator area starting at address >834A. The number will be in Radix 100 notation.

[STRREF] STRING REFERENCE

This utility allows you to get a string that was passed via CALL LINK command from BASIC. You must reserve an area of memory to hold the string before calling this utility. The following steps outline how this accomplished:

1. Reserve a buffer area in memory to hold the string. The first byte of the buffer area should hold the length of the string. If the the string length actually exceeds this value, an error is generated. Otherwise the actual length is placed in the first byte.

2. Place 0 in R0 if it is a simple string variable. Place the element number if the string is in an array.

3. Load the starting address of the buffer in R2.

4. Call the utility.

ERROR REPORTING

This utility allows you to transfer control to the error reporting routine in BASIC. To use this utility all you have to do is load the error code into the most significant byte of R0 and call the utility via BLWP @ERR or BLWP @6050.

The error codes that can be listed by your program are found in Table 13.3 on the adjacent page.

TABLE 12.3 BASIC ERROR CODES

CODE	ERROR MESSAGE	CODE	ERROR MESSAGE
00	I/O error (bad name)	14	Number too big
01	I/O error (write protected)	15	String-number mismatch
02	I/O error (bad attribute)	16	Bad argument
03	I/O error (illegal operation)	17	Bad subscript
04	I/O error (buffer full)	18	Name conflict
05	I/O error (read past EOF)	19	Can't do that
06	I/O error (device error)	1A	Name conflict
07	I/O error (file error)	1B	For-Next error
08	Memory full (closes file)	1C	I/O error
09	N/A	1D	File error
0A	Bad tag	1E	Input error
0B	Checksum error	1F	Data error
0C	Duplicate definition	20	Line too long
0D	Unresolved references	21	Memory full (file not closed)
0E	N/A		
0F	Program not found	22	Syntax error
10	Incorrect statement	23	Numeric overflow
11	Bad name	24	Unrecognized character
12	Can't continue	25	String truncated
13	Bad value	26-FF	Unknown error

13

HIGH PRECISION MATHEMATICS

Along with the many utilities discussed in Chapter 6, there are many additional utility programs related to mathematics that literally save you hours (or days) in programming time.

The first section of this chapter outlines mathematical GPL routines that can be accessed through GPLLNK. The second section of this chapter discusses ROM console routines that can be accessed through XMLLNK.

All of the following routines involve floating point numbers. If an error occurs during execution of the routine, the error is indicated in byte >8345. Table 13.0 gives all the possible error codes that can be returned.

TABLE 13.0 FLOATING POINT ROUTINE ERROR CODES

CODE	ERROR TYPE
>01	Overflow.
>02	Syntax error.
>03	Integer overflow on conversion.
>04	Square root of a negative number.
>05	Negative number to non-integer power.
>06	Logarithm of a non-positive number.
>07	Invalid argument in trigonometric fxn.

Table 13.1 outlines the mathematical routines that can be accessed through GPLLNK.

TABLE 13.1 XML ROUTINE CODES

ROUTINE CODE	DESCRIPTION
>0014	Convert number to string.
>0022	Greatest integer function.
>0024	Involution routine.
>0026	Square root routine.
>0028	Exponent routine.
>002A	Natural logarithm routine.
>002C	Cosine routine.
>002E	Sine routine
>0030	Tangent routine.
>0032	Arctangent.

The sections that follow in this chapter describe the GPL mathematical routines. The address of the Floating Point Accumulator is >834A. The Floating Point Accumulator is abbreviated FAC in the following sections.

Parentheses indicates the BASIC statement which would call the routine from a BASIC program.

DATA >0014 [STR] CONVERT NUMBER TO STRING

This routine allows you to convert a floating point number into a ASCII string. The following are the necessary steps:

1. The eight bytes defining the number are located beginning at FAC.

2. If you set FAC+11 (>8355) equal to zero, it indicates that the output string is to be in BASIC format. Otherwise the output is in FIX mode, which requires data in FAC+12 and FAC+13 (>8356 & >8357).

   FAC+12 is the number of significant bytes. If 1, it expresses overflow from the calculation range.

   FAC+13 indicates the number of digits to the right of the decimal point. A negative value disables the FIX mode.

3. After the execution of the STR routine, FAC is modified. FAC+11 (>8355) contains the least significant byte of the address where the string is located. This byte must be added to >8300 to find the actual address of the string; address=(FAC+11)+>8300. FAC+12 (>8356) contains the length of the string (in bytes).

DATA >0022 [INT] GREATEST INTEGER FUNCTION

This routine allows you to compute the greatest integer contained in a value.

1. FAC contains the floating point value.

2. After calling this routine, FAC contains the result. For positive numbers, the integer is the truncated value. For negative numbers, the integer is the truncated value plus one.

3. The GPL status byte (>837C) is set according to the result.

DATA >0024 INVOLUTION ROUTINE

This routine allows you to raise a number to a specified power.

1. FAC contains the exponent value.

2. Address >836E (STACK) contains the address in VDP RAM that holds the eight byte number.

3. The result is placed in FAC in floating-point format. This is computed as exp*LOG[ABS(base)].

4. After completion of this routine, the data at addresses >8375 and >8376 is destroyed. The word at address >836E is decremented by 8.

DATA >0026 [SQR] SQUARE ROOT ROUTINE

This routine allows you to find the square root of a number.

1. FAC contains the input value.

2. After the routine, FAC contains the square root of the input value.

3. The GPL status byte is affected.

4. Addresses >8375 and >8376 are destroyed by this routine.

DATA >0028 [EXP] EXPONENT ROUTINE

This routine will compute the inverse natural logarithm of a number.

1. FAC contains the input value.

2. After the routine, FAC contains the resulting value.

3. The GPL status byte is affected.

4. Addresses >8375 and >8376 are destroyed by this routine.

DATA >002A [LOG] NATURAL LOGARITHM ROUTINE

This routine will compute the natural logarithm of a number.

1. FAC contains the input value.

2. After the routine, FAC contains the resulting value.

3. The GPL status byte is affected.

4. Addresses >8375 and >8376 are destroyed by this routine.

DATA >002C [COS] COSINE ROUTINE

This routine will compute the cosine of a number that is expressed in radians.

1. FAC contains the input value.
2. After the routine, FAC contains the cosine of the input value.
3. The GPL status byte is affected.
4. Addresses >8375 and >8376 are destroyed by this routine.

DATA >002E [SIN] SINE ROUTINE

This routine will compute the sine of a number expressed in radians.

1. FAC contains the input value.
2. After the routine, FAC contains the sine of the input value.
3. The GPL status byte is affected.
4. Addresses >8375 and >8376 are destroyed by this routine.

DATA >0030 [TAN] TANGENT ROUTINE

This routine will compute the tangent of a number expressed in radians.

1. FAC contains the input value.
2. After the routine, FAC contains the tangent of the input value.
3. The GPL status byte is affected.
4. Addresses >8375 and >8376 are destroyed by this routine.

DATA >0032 [ARC] ARCTANGENT ROUTINE

This routine will compute the arctangent of a number expressed in radians.

1. FAC contains the input value.

2. After the routine, FAC contains the arctangent off the input value.

3. The GPL status byte is affected.

4. Addresses >8375 and >8376 are destroyed by this routine.

To review how to call up GPL routines through the use of the GPLLNK utility, refer to page 82 of chapter 6. Remember that you must reset the GPL status byte at address >837C, or a meaningless error message will be returned. Also make sure that any of the CPU RAM areas that are affected by a GPL routine are not being used by your program to store information. The addresses that you need to use these utilities with the mini memory module can be found in table 10.1 on page 148.

Routines that are located in ROM can be accessed through the use of the XMLLNK command.

There are two ways to access a routine in console ROM. The first is to specify the routine's code in a DATA statement. For example,

```
BLWP  @XMLLNK
DATA  >0800
```

branches to the floating-point multiplication routine in the console.

The second way to access a routine in console ROM is to specify its addresses in the DATA statement. You should take note that when using this method, the most significant bit of the DATA word must be set to indicate to the system that this is an address instead of a routine code. For example,

```
BLWP  @XMLLNK   * 8  D  3  A     (note MSB set to indicate
DATA  >8D3A     * 1000  1101  0011  1010       an address)
```

branches to console ROM address >0D3A which is the floating point compare routine.

Unless absolutely unavoidable, you should not use direct memory addresses of console ROM routines as they can vary from one console to another. Table 13.2 outlines the console routine codes that can be used with XMLLNK.

TABLE 14.2 XML ROUTINES

Routine Code	Description
>0600	Floating-Point Addition
>0700	Floating-Point Subtraction
>0800	Floating-Point Multiplication
>0900	Floating-Point Division
>0A00	Floating-Point Compare Operation
>0B00	Floating-Point Stack Addition
>0C00	Floating-Point Stack Subtraction
>0D00	Floating-Point Stack Multiplication
>0E00	Floating-Point Stack Division
>0F00	Floating-Point Stack Comparison
>1000	Convert String to Number
>1200	Convert Floating-Point to Integer
>1700	Push a value onto Value Stack
>1800	Pop a Value for the Value Stack
>1230	Convert Integer to Floating-Point

In the routines that follow, FAC starts at address >834A, ARG (which stands for arguments) starts at address >835C. STACK is at address >836E.

All overflow errors, except in convert floating point to integer, return >01 at address >8354.

DATA >0600 FLOATING POINT ADDITION

This routine adds two values.

1. FAC contains the first value..
2. ARG contains the second value.
3. FAC holds the result after calling the routine.

DATA >0700 FLOATING POINT SUBTRACTION

This routine subtracts two values.

1. FAC contains the value to be subtracted.
2. ARG contains the value from which FAC is subtracted.
3. FAC holds the result of the subtraction after calling the routine.

DATA >0800 FLOATING POINT MULTIPLICATION

This routine multiplies two numbers together.

1. FAC holds the value of the multiplier.
2. ARG holds the value of the multiplicand.
3. FAC holds the result after the routine is called.

DATA >0900 FLOATING POINT DIVISION

This routine divides two values.

1. FAC holds the divisor.
2. ARG holds the dividend.
3. FAC holds the result of the operation after calling the utility.

DATA >0A00 FLOATING POINT COMPARE

This routine compares two floating point numbers.

1. FAC holds the first number while ARG holds the second.
2. The GPL status byte (>837C) is affected. The high bit is set if ARG is logically higher than FAC. The greater than bit is set if ARG is arithmetically higher than FAC. The equal bit is set if ARG and FAC are equal.

DATA >0B00 VALUE STACK ADDITION

This routine will add using a stack in VDP RAM.

1. STACK contains the VDP RAM address where the left-hand term is located.
2. FAC holds the right-hand term.
3. FAC holds the result of the addition after the addition after the routine is called.

DATA >0C00 VALUE STACK SUBTRACTION

This routine will subtract using a stack in VDP RAM.

1. STACK contains the VDP RAN address of the multiplicand.
2. FAC contains the multiplier.
3. FAC holds the result of the multiplication after calling the routine.

DATA >0D00 VALUE STACK MULTIPLICATION

This routine will multiply using a stack in VDP RAM.

1. Stack contains the VDP RAM address of the multiplicand.
2. FAC contains the multiplier.
3. FAC holds the result of the multiplication after the routine has been called.

DATA >0E00 VALUE STACK DIVISION

This routine will divide using a stack in VDP RAM

1. STACK contains the VDP RAM address holds the dividend.
2. FAC holds the divisor value.
3. FAC holds the result of the division after the routine has been called.

DATA >0F00 VALUE STACK COMPARE

This routine will compare a value in the VDP RAM stack to the value in FAC.

1. STACK holds the VDP RAM address of the value to be compared.
2. FAC holds the other value to be compared.

3. The GPL status byte (>837C) is affected. The high bit is set if STACK is logically higher than FAC. The greater than bit is set if STACK is arithmetically higher than FAC. The equal bit is set if STACK and FAC are equal.

DATA >1000 CONVERT STRING TO NUMBER

This routine will convert an ASCII string into a floating-point number.

1. FAC+12 (>8356) is the address of the starting in VDP RAM.

2. FAC holds the result of the conversion in floating-point format.

DATA >1200 CONVERT FLOATING POINT TO INTEGER

This routine will convert a floating-point number into an integer.

1. FAC contains the floating-point number to be converted.

2. FAC will contain integer value as one word. The maximum value of this word is >FFFF. If there is an overflow, FAC+10 (>8354) is set to the overflow error code, >03.

DATA >1700 PUSH VALUE ONTO VALUE STACK

This routine will push a value you have loaded in FAC onto the value stack.

DATA >1800 POP VALUE FROM VALUE STACK

This routine will pop a value from the value stack and place it in FAC.

1. FAC contains the one-word integer that is to be converted.

2. FAC will contain the floating-point result after the routine is called.

NOTE: This routine is only available with the Editor/Assembler and is not supported in Extended Basic or by the Line-by-Line assembler. It has also been found that the correct code for this routine may be >7200 in some consoles.

INDEX

APPENDIX A

HEXADECIMAL/DECIMAL INTERCONVERSIONS

6		5		4		3		2		1	
HX	DEC	HX	DEC	HX	DEC	HX	DEC	HX	DEC	HX	DEC
0	0	0	0	0	0	0	0	0	0	0	0
1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	15,728,640	F	983,040	F	61,440	F	3,840	F	240	F	15

HX=hexadecimal DEC=decimal

POWERS OF 2

2 x	x
1	0
2	1
4	2
8	3
16	4
32	5
64	6
128	7
256	8
512	9
1024	10
2048	11
4096	12
8192	13
16,384	14
32,768	15
65,536	16
131,072	17

POWERS OF 16

16 x	x
1	0
16	1
256	2
4,096	3
65,536	4
1,048,576	5
16,777,216	6
268,435,456	7
4,294,967,296	8
68,719,476,736	9
1,099,511,627,776	10
17,592,186,044,416	11
281,474,976,710,656	12
4,503,599,627,370,496	13
72,057,594,037,927,936	14
1,152,921,504,606,846,976	15

TMS9900 INSTRUCTION SET SUMMARY

The following table summarizes the TMS9900 instruction set in alphabetical order. For each instruction it shows the general assembler format and which status flags are affected (x) or tested (t). The page number given refers to where in the book the instruction is described.

THE TMS9900 INSTRUCTION SET

Mnemonic	Format	Page	Status Flags										
			L>	A>	EQ	C	OV	OP	X	Int		Mask	
A	G,(G)	36	x	x	x	x	x	-	-	-	-	-	-
AB	G,(G)	36	x	x	x	x	x	x	-	-	-	-	-
ABS	G	37	x	x	x	-	x	-	-	-	-	-	-
AI	(W),#	37	x	x	x	x	x	-	-	-	-	-	-
ANDI	(W),#	50	x	x	x	-	-	-	-	-	-	-	-
B	G	43	-	-	-	-	-	-	-	-	-	-	-
BL	G	43	-	-	-	-	-	-	-	-	-	-	-
BLWP	G	43	-	-	-	-	-	-	-	-	-	-	-
C	G,G	47	x	x	x	-	-	-	-	-	-	-	-
CB	G,G	48	x	x	x	-	-	x	-	-	-	-	-
CI	W,#	48	x	x	x	-	-	-	-	-	-	-	-
CLR	G	54	-	-	-	-	-	-	-	-	-	-	-
COC	G,W	48	-	-	x	-	-	-	-	-	-	-	-
CZC	G,W	49	-	-	x	-	-	-	-	-	-	-	-
DEC	G	37	x	x	x	x	x	-	-	-	-	-	-
DECT	G	38	x	x	x	x	x	-	-	-	-	-	-
DIV	G,(W)	38	-	-	-	-	x	-	-	-	-	-	-

THE TMS9900 INSTRUCTION SET (continued)

Mnemonic	Format	Page	Status Flags										
			L>	A>	EQ	C	OV	OP	X	Int Mask			
INC	G	39	x	x	x	x	x	-	-	-	-	-	-
INCT	G	40	x	x	x	x	x	-	-	-	-	-	-
INV	G	53	x	x	x	-	-	-	-	-	-	-	-
JEQ	expression	46	-	-	t	-	-	-	-	-	-	-	-
JGT	expression	46	-	t	-	-	-	-	-	-	-	-	-
JH	expression	46	t	-	t	-	-	-	-	-	-	-	-
JHE	expression	46	t	-	t	-	-	-	-	-	-	-	-
JL	expression	46	t	-	t	-	-	-	-	-	-	-	-
JLE	expression	46	t	-	t	-	-	-	-	-	-	-	-
JLT	expression	46	-	t	t	-	-	-	-	-	-	-	-
JMP	expression	44	-	-	-	-	-	-	-	-	-	-	-
JNC	expression	46	-	-	-	t	-	-	-	-	-	-	-
JNE	expression	46	-	-	t	-	-	-	-	-	-	-	-
JNO	expression	46	-	-	-	-	t	-	-	-	-	-	-
JOC	expression	46	-	-	-	t	-	-	-	-	-	-	-
JOP	expression	46	-	-	-	-	-	t	-	-	-	-	-
LI	(W),#	33	x	x	x	-	-	-	-	-	-	-	-
LIMI	#	34	-	-	-	-	-	-	-	x	x	x	x
LWPI	#	34	-	-	-	-	-	-	-	-	-	-	-
MOV	G,(G)	33	x	x	x	-	-	-	-	-	-	-	-
MOVB	G,(G)	33	x	x	x	-	-	x	-	-	-	-	-
MPY	G,(W)	40	-	-	-	-	-	-	-	-	-	-	-
NEG	G	40	x	x	x	-	x	-	-	-	-	-	-
NOP		61	-	-	-	-	-	-	-	-	-	-	-
ORI	(W),#	50	x	x	x	-	-	-	-	-	-	-	-

THE TMS9900 INSTRUCTION SET (continued)

Mnemonic	Format	Page	Status Flags										
			L>	A>	EQ	C	OV	OP	X	Int		Mask	
RT		62	-	-	-	-	-	-	-	-	-	-	-
RTWP		44	×	×	×	×	×	×	×	×	×	×	×
S	G,(G)	41	×	×	×	×	×	-	-	-	-	-	-
SB	G,(G)	41	×	×	×	×	×	×	-	-	-	-	-
SETO	G	54	-	-	-	-	-	-	-	-	-	-	-
SLA	(W),#	57	×	×	×	×	×	-	-	-	-	-	-
SOC	G	54	×	×	×	-	-	-	-	-	-	-	-
SOCB	G,(G)	55	×	×	×	-	-	×	-	-	-	-	-
SRA	(W),#	57	×	×	×	×	-	-	-	-	-	-	-
SRL	(W),#	59	×	×	×	×	-	-	-	-	-	-	-
SRC	(W),#	59	×	×	×	×	-	-	-	-	-	-	-
STST	W	34	-	-	-	-	-	-	-	-	-	-	-
STWP	W	34	-	-	-	-	-	-	-	-	-	-	-
SWPB	G	35	-	-	-	-	-	-	-	-	-	-	-
SZC	G,(G)	55	×	×	×	-	-	-	-	-	-	-	-
SZCB	G,(G)	56	×	×	×	-	-	×	-	-	-	-	-
X	G	46	×	×	×	×	×	×	×	×	×	×	×
XOP	G,(W)	31	-	-	-	-	-	-	-	-	-	-	-
XOR	G,(W)	50	×	×	×	-	-	-	-	-	-	-	-

APPENDIX C

THE ASSEMBLER DIRECTIVE COMMAND SET

The following table summarizes the Editor/Assembler director set. These directives are in alphabetical order. For each directive is shown the general assembler format. The page number given refers to where in the book the directive is described, parentheses indicate where the directive is described when referring to source code created using the mini memory module.

DIRECTIVE COMMAND SET

Mnemonic	Format	Page #
AORG	word[expression]	67(138)
BES	word[expression]	68
BSS	word[expression]	67(138)
BYTE	exp,exp...expression	70
COPY	"File-name"	72
DATA	exp,exp...expression	70(138)
DEF	symbol,symbol...symbol	72
DORG	expression	67
END	Symbol	73(141)
EQU	expression	69(138)
EVEN		68
IDT		74
LIST		74
PAGE		75
REF	symbol,symbol...symbol	72
RORG	expression	67
SYM		(139)
TEXT	'string'	71(138)
TITL	'string'	75
UNL		74
XOP	Symbol,term	73

APPENDIX D

This appendix contains a few source code listings that may be of interest. These examples are of short game program modules that you can incorporate into your programs. Where possible, The BASIC version of the program is presented for comparison purposes.

The first program module sets a red ball-shaped sprite in motion only when the joystick is moved. The border color is black but the main screen is left uncolored (light green is the default color).

```
10 CALL CLEAR
20 CALL CHAR(80,"3C7EFFFFFFFF7E3C")
30 CALL SPRITE(#1,80,9,100,100)
40 CALL JOYST(1,X,Y)
50 CALL MOTION(#1,-Y*4,X*4)
60 GOTO 40
```

Notice that the source code listing for the same program is much longer. This allows you much greater flexability, but the price is much more time spent programming.

```
          DEF   SPRITE
          REF   VMBW,VWTR,KSCAN
*
JOY1      BYTE  4,0
JOY2      BYTE  4,4
JOY3      BYTE  0,4
JOY4      BYTE  -4,4
JOY5      BYTE  -4,0
JOY6      BYTE  -4,-4
JOY7      BYTE  0,-4
JOY8      BYTE  4,-4
ONE       BYTE  1
          EVEN
*
JOYY      EQU   >8376
KBOARD    EQU   >8374
NUMB      EQU   >837A
SATAB     EQU   >0300
SDTAB     EQU   >0400
SMTAB     EQU   >0780
*
BALL      DATA  >3C7E,>FFFF,>FFFF,>7E3C
SDATA     DATA  >70D0,>8008
          DATA  >D000
*
SPO       DATA  >0000,>0000
*
```

```
MYREG   BSS   >20
SPRITE  LWPI  MYREG
        LI    R0,SDTAB
        LI    R1,BALL
        LI    R2,8
        BLWP  @VMBW
*
        LI    R0,SATAB
        LI    R1,SDATA
        LI    R2,8
        BLWP  @VMBW
*
        LI    R1,1
        SLA   R1,8
        MOVB  R1,@NUMB
*
        LI    R0,>0701
        BLWP  @VWTR
*
        MOVB  @ONE,@KBOARD
*
LOOP    LI    R0,SMTAB
        LI    R1,SPO
        LI    R2,4
        BLWP  @VMBW
*
LOOP1   LIMI  2
        LIMI  0
        BLWP  @KSCAN
        MOV   @JOYY,@JOYY
        JEQ   LOOP
*
        C     @JOYY,@JOY1
        JEQ   M1
        C     @JOYY,@JOY2
        JEQ   M2
        C     @JOYY,@JOY3
        JEQ   M3
        C     @JOYY,@JOY4
        JEQ   M4
        C     @JOYY,@JOY5
        JEQ   M5
        C     @JOYY,@JOY6
        JEQ   M6
        C     @JOYY,@JOY7
        JEQ   M7
*
```

```
          LI    R1,JOY6
          B     @CHANGE
M1        LI    R1,JOY5
          B     @CHANGE
M2        LI    R1,JOY4
          B     @CHANGE
M3        LI    R1,JOY3
          B     @CHANGE
M4        LI    R1,JOY2
          B     @CHANGE
M5        LI    R1,JOY1
          B     @CHANGE
M6        LI    R1,JOY8
          B     @CHANGE
M7        LI    R1,JOY7
*
CHANGE    LI    R0,SMTAB
          LI    R2,2
          BLWP  @VMBW
*
          B     @LOOP1
          END
```

The last program worked well enough but it went about it the long way around. Using a little ingenuity we can considerably shorthen the above program. The source listing that follows accomplishes the same task as the last program, only it has been shortened with some programming tricks.

```
          DEF   SPRITE
          REF   VMBW,VWTR,KSCAN
*
ONE       BYTE  1
ZERO      BYTE  2
*
JOYY      EQU   >8376
KBOARD    EQU   >8374
NUMB      EQU   >837A
SATAB     EQU   >0300
SDTAB     EQU   >0400
SMTAB     EQU   >0780
*
BALL      DATA  >3C7E,>FFFF,>FFFF,>7E3C
SDATA     DATA  >70D0,>8008
          DATA  >0000
*
SPD       DATA  >0000,>0000
*
```

```
MYREG   BSS   >20
SPRITE  LWPI  MYREG
        LI    R0,SDTAB
        LI    R1,BALL
        LI    R2,8
        BLWP  @VMBW
*
        LI    R0,SATAB
        LI    R1,SDATA
        LI    R2,8
        BLWP  @VMBW
*
        LI    R1,1
        SLA   R1,8
        MOVB  R1,@NUMB
*
LOOP    LI    R0,SMTAB
        LI    R1,SPO
        LI    R2,4
        BLWP  @VMBW
*
LOOP1   LIMI  2
        LIMI  0
        BLWP  @KSCAN
        MOV   @JOYY,@JOYY
        JEQ   LOOP
        CB    @JOYY,@ZERO
        JEQ   GO
        MOVB  @JOY+1,@R5
        NEG   @JOYY
        MOVB  R5,@JOYY+1
        LI    R1,JOYY
        JMP   CHANGE
*
GO      LI    R1,JOYY
CHANGE  LI    R0,SMTAB
        LI    R2,2
        BLWP  @VMBW
*
        B     @LOOP1
        END
```

The last two programs can be loaded via the LOAD AND RUN option of the Editor/Assembler and run by typing in SPRITE in response to the PROGRAM NAME? prompt. In order to run these programs using the Mini Memory module you must:

1. Alter the length of all LABEL fields to two characters.

2. Use appropriate address instead of symbols for the utility programs.

3. Enter the program name and starting point into the REF/DEF table (refer to page 145).

The third program in this series illustrates additive motion. The longer you hold the joystick in one direction, the faster your sprite will move (here a red ball again!) To stop the sprite you will have to cancel out the motion by holding the joystick in the opposite direction to "brake" the sprite. This module lends itself well to incorporation of "space games" where you have to simulate the abscence of gravity.

```
10  CALL CLEAR
20  CALL SCREEN(2)
30  CALL CHAR(128,"3C7EFFFFFFFF7E3C")
40  CALL SPRITE(#1,128,9,100,100)
50  CALL JOYST(1,C,R)
60  X=(X+C)*-(ABS(X)<124)
70  Y=(Y-R)*-(ABS(Y)<124)
80  CALL MOTION(#1,Y,X)
```

NOTE: A direct translation of line 60 is:

IF THE ABSOLUTE VALUE OF X<124 THEN X=X+C ELSE X=0

The following program is the assembly language version of the above program. Note how the "CALL SCREEN" code was added.

```
        DEF   SPRITE
        REF   VMBW,VWTR,KSCAN,VSBW
*
JOY1    BYTE  4,0
JOY2    BYTE  4,4
JOY3    BYTE  0,4
JOY4    BYTE  -4,4
JOY5    BYTE  -4,0
JOY6    BYTE  -4,-4
JOY7    BYTE  0,-4
JOY8    BYTE  4,-4
ONE     BYTE  1
        EVEN
*
```

```
JOYY    EQU   >8376
KBOARD  EQU   >8374
NUMB    EQU   >837A
COLTAB  EQU   >0384
SATAB   EQU   >0300
SDTAB   EQU   >0400
SMTAB   EQU   >0780
*
BALL    DATA  >3C7E
SDATA   DATA  >70D0,>8008
        DATA  >D000
SPEED   DATA  >0000,>0000
COLOR   DATA  >1000
*
SPRITE  LWPI  MYREG
        LI    R0,COLTAB
        MOV   @COLOR,R1
        BLWP  @VSBW
*
        CLR   R0
        LI    R1,>2000
        LI    R2,767
LOOP    BLWP  @VSBW
        INC   R0
        DEC   R2
        JGT   LOOP
*
        LI    R0,>0701
        BLWP  @VWTR
*
        LI    R0,SATAB
        LI    R1,SDATA
        LI    R2,8
        BLWP  @VMBW
*
        LI    R0,SDTAB
        LI    R1,BALL
        LI    R2,8
        BLWP  @VMBW
*
        LI    R1,1
        SLA   R1,8
        MOVB  R1,@NUMB
*
        MOVB  @ONE,@KBOARD
        LI    R0,SMTAB
        LI    R1,SPEED
        LI    R2,4
        BLWP  @VMBW
*
```

```
LOOP1   LIMI  2
        LIMI  0
        BLWP  @KSCAN
        MOV   @JOYY,@JOYY
        JEQ   LOOP1
*
        C     @JOYY,@JOY1
        JEQ   UP
        C     @JOYY,@JOY3
        JEQ   RIGHT
        C     @JOYY,@JOY5
        JEQ   DOWN
        C     @JOYY,@JOY7
        JEQ   LEFT
        C     @JOYY,@JOY2
        JEQ   UPRT
        C     @JOYY,@JOY4
        JEQ   DNRT
        C     @JOYY,@JOY6
        JEQ   DNLT
        C     @JOYY,@JOY8
        JEQ   UPLT
        JMP   LOOP1
*
UP      DECT  R5
        B     @ADJUST
UPRT    DECT  R5
        INCT  R6
        B     @ADJUST
RIGHT   INCT  R6
        B     @ADJUST
DNRT    INCT  R5
        INCT  R6
        B     @ADJUST
DOWN    INCT  R5
        B     @ADJUST
DNLT    INCT  R5
        DECT  R6
        B     @ADJUST
LEFT    DECT  R6
        B     @ADJUST
UPLT    DECT  R5
        DECT  R6
*
ADJUST  LI    R0,SMTAB
        MOVB  R5,R1
        BLWP  @VSBW
*
        LI    R0,SMTAB+1
        MOVB  R6,R1
        BLWP  @VSBW
        B     LOOP
        END
```

This next program will illustrate the double-size and m,agnified sprite concept. When the program is run the sprite (a red ball) is standard sized and in motion across the screen. When any key is pressed the sprite is doublesized. When a key is pressed again the sprite is magnified. Finally, a third press of a key will make the sprite double-sized and magnified. Subsequent pressings of a key will repeat the cycle.

```
          DEF   SPRITE
          REF   VMBW,VSBR,VSBW,KSCAN,VWTR
*
KBOARD    EQU   >8375
SKEY      EQU   >8374
SATAB     EQU   >0300
SDATA     EQU   >0400
*
BALL      DATA  >3C7E,>FFFF,>FFFF,>7E3C
SDATA     DATA  >7080,>8008
*
STATUS    EQU   >837C
SET       DATA  >2000
MYREG     BSS   >20
*
SPRITE    LWPI  MYREG
          LI    R3,4
          LI    R0,SDTAB
START     LI    R1,BALL
          LI    R2,8
          BLWP  @VMBW
          AI    R0,8
          DEC   R3
          JNE   START
*
          LI    R0,SATAB
          LI    R1,SDATA
          LI    R2,6
          BLWP  @VMBW
*
LOOP      LI    R0,SATAB+1
READ      BLWP  @VSBR
          SRL   R1,8
          DEC   R1
          JNE   MOVE
          LI    R1,>00FF
*
MOVE      SLA   R1,8
          BLWP  @VSBW
          CLR   R8
DELAY     INC   R8
          CI    R8,800
          JNE   DELAY
```

```
*
OUT     CLR   @KBOARD
        BLWP  @KSCAN
        MOV   @STATUS,R3
        COC   @SET,R3
        JNE   LOOP
*
CHECK   INC   R6
        CI    R6,4
        JLT   GO
        CLR   R6
*
GO      CI    R6,1
        JEQ   MAG
        CI    R6,2
        JEQ   DSIZE
        CI    R6,3
        JEQ   DSIZEM
*
SMALL   LI    R0,>01E0
        JMP   WRITE
MAG     LI    R0,>01E1
        JMP   WRITE
        LI    R0,>01E2
        JMP   WRITE
DSIZEM  LI    R0,>01E3
        JMP   WRITE
*
WRITE   BLWP  @VWTR
        B     @LOOP
        END
```

Can you sum
all this code
into a two
line
statement?

The next program places six red ball-shaped characters (not sprites) on the screen in a diagonal pattern. The screen is then scrolled upwards. This type of motion is familiar to anybody who has played the Alpiner game from TI. In oreder to run this program you must type GRAPH in response to the PROGRAM NAME? prompt.

```
        DEF   GRAPH
        REF   VSBW,VMBW,VMBR
*
BALL    DATA  >3C7E,>FFFF,>FFFF,>7E3C
COLOR   DATA  >8100
*
COLTAB  EQU   >0384
PATTAB  EQU   >0908
*
MYREG   BSS   >20
*
GRAPH   LWPI  MYREG
        LI    R0,COLTAB
```

```
            MOV   @COLOR,R1
            BLWP  @VMBW
*
            LI    R0,PATTAB
            LI    R1,BALL
            LI    R2,8
            BLWP  @VMBW
*
            LI    R0,325
            LI    R1,>2100
            LI    R2,6
LOOP        BLWP  @VMBW
            AI    R0,33
            DEC   R2
            JGT   LOOP
*
LINE1       BSS   32
LINEX       BSS   32
*
SCROLL      CLR   R0
            LI    R1,LINE1
            LI    R2,>20
            BLWP  @VMBR
*
            LI    R0,>20
            LI    R1,LINEX
            LI    R2,>20
            BLWP  @VMBR
*
            LI    R0,0
LOOP1       BLWP  @VMBW
            AI    R0,>40
            CI    R0,>300
            JHE   OUT
            BLWP  @VMBR
            AI    R0,>FFE0
            JMP   LOOP1
*
OUT         LI    R0,>2E0
            LI    R1,LINE1
            BLWP  @VMBW
            JMP   SCROLL
            END
*
OUT         CB    @KEYUP,@KEY
            JEQ   PUP
            CB    @KEYDN,@KEY
            JEQ   PDOWN
            CB    @KEYRT,@KEY
            JEQ   PRIGHT
            CB    @KEYLT,@KEY
```

```
        JEQ   PLEFT
        JMP   START
*
PUP     LI    R1,UP
        JMP   PRINT
PDOWN   LI    R1,DOWN
        JMP   PRINT
PRIGHT  LI    R1,RIGHT
        JMP   PRINT
PLEFT   LI    R1,LEFT
        JMP   PRINT
*
PRINT   LI    R0,325
        LI    R2,17
        BLWP  @VMBW
        JMP   START
        END   BEGIN
```

APPENDIX E

INPUT/OUTPUT ERROR CODES

Error code	Meaning
0	Bad device name.
1	Device is write protected.
2	Bad open attribute. 1. incorrect file type. 2. incorrect record length 3. incorrect I/O mode 4. no records in relative record file
3	Illegal operation. 1. conflict with OPEN attributes 2. peripheral does not support operation
4	Out of buffer space on device.
5	Attempt to read past end of file. File is closed.
6	Device error. Mechanical or medium failure.
7	File error. 1. program/data mismatch 2. openning non-existing file in INPUT

APPENDIX F

EXECUTION-TIME ERRORS

This table lists errors that may be generated when you attempt to run your program.

Error code	Meaning
00-07	Input/Output error
08	MEMORY FULL
09	INCORRECT STATEMENT
0A	ILLEGAL TAG
0B	CHECKSUM ERROR
0C	DUPLICATE DEFINITION
0D	UNRESOLVED REFERENCE
0E	INCORRECT STATEMENT
0F	PROGRAM NOT FOUND
10	INCORRECT STATEMENT
11	BAD NAME
12	CAN'T CONTINUE
13	BAD VALUE
14	NUMBER TOO BIG
15	STRING-NUMBER MISMATCH
16	BAD ARGUEMENT
17	BAD SUBSCRIPT
18	NAME CONFLICT
19	CAN'T DO THAT
1A	BAD LINE NUMBER
1B	FOR-NEXT ERROR
1C	Input/Output error
1D	FILE ERROR
1E	INPUT ERROR
1F	DATA ERROR
20	LINE TOO LONG
21	MEMORY FULL
22-FF	Unknown error code

ANSWERS TO CHAPTER QUESTIONS

CHAPTER 2

1. a) 1111
   b) 0001 1000
   c) 0111 0101 1010 1001
   d) 1101 0111 1111 0110

2. a) 4
   b) 9453

3. a) >F
   b) >18
   c) >75A9
   d) >D7F6

4. a) 7220
   b) 7220

CHAPTER 3

1. Assembler program

2. Sixteen

3. Overflow bit (OF)

4. a) Label field, Comment field, Operand field
   b) (*) asterisk
   c) Op-code field (operation code)

5. Assembler directives give instructions to the assembler program as to what to do with program instructions while the program instructions make up the actual object code.

6. NO (G is not a HEX character)

7. Op-code field (operation code)

CHAPTER 4

1.
```
* R3 CONTAINS VALUE
      MOV R3,R4
      SLA R3,2
      SLA R4,3
      A   R3,R4
```

2. Workspace Register 11 (R11)

3. 256 bytes (>100)

4. Conditional jump instructions

5. Return (RT)

6. a) WR,WR-indirect
   b) WR-autoincrement,WR
   c) Symbolic,Symbolic
   d) WR,Indexed

CHAPTER 5

1. a) No such instruction as 'AND'
   b) No such instruction as 'OR'
   c) >5756
   d) >126C
   e) >48E8

2. It is an endless loop because R3 is constantly re-loaded and never will be equal to zero.

3.
```
SAVE  DATA  >0000
      MOV   R3,@SAVE
```

4. Nothing, MPY requires two operands.

CHAPTER 6

1.
```
      LI    R0,>1000
      LI    R1,>2200
      BLWP  @VSBW
```

2. No key has been pressed.

3.
```
      MOVB  >01,>8374
```

CHAPTER 7

1.
```
      REF   VWTR
      .
      LI    R0,>01E8
      BLWP  @VWTR
```

2.
```
      REF   VWTR
      .
      LI    R0,>01EB
      BLWP  @VWTR
```

3. Change the value of VDP register 1 to >01.

4. Review pages 117 to 125.

5. The computer views the screen as a series of memory locations in VDP RAM numbered >000 through >767.

6. Place the entry point of the program in a END directive. The program will begin running as soon as it is entered.