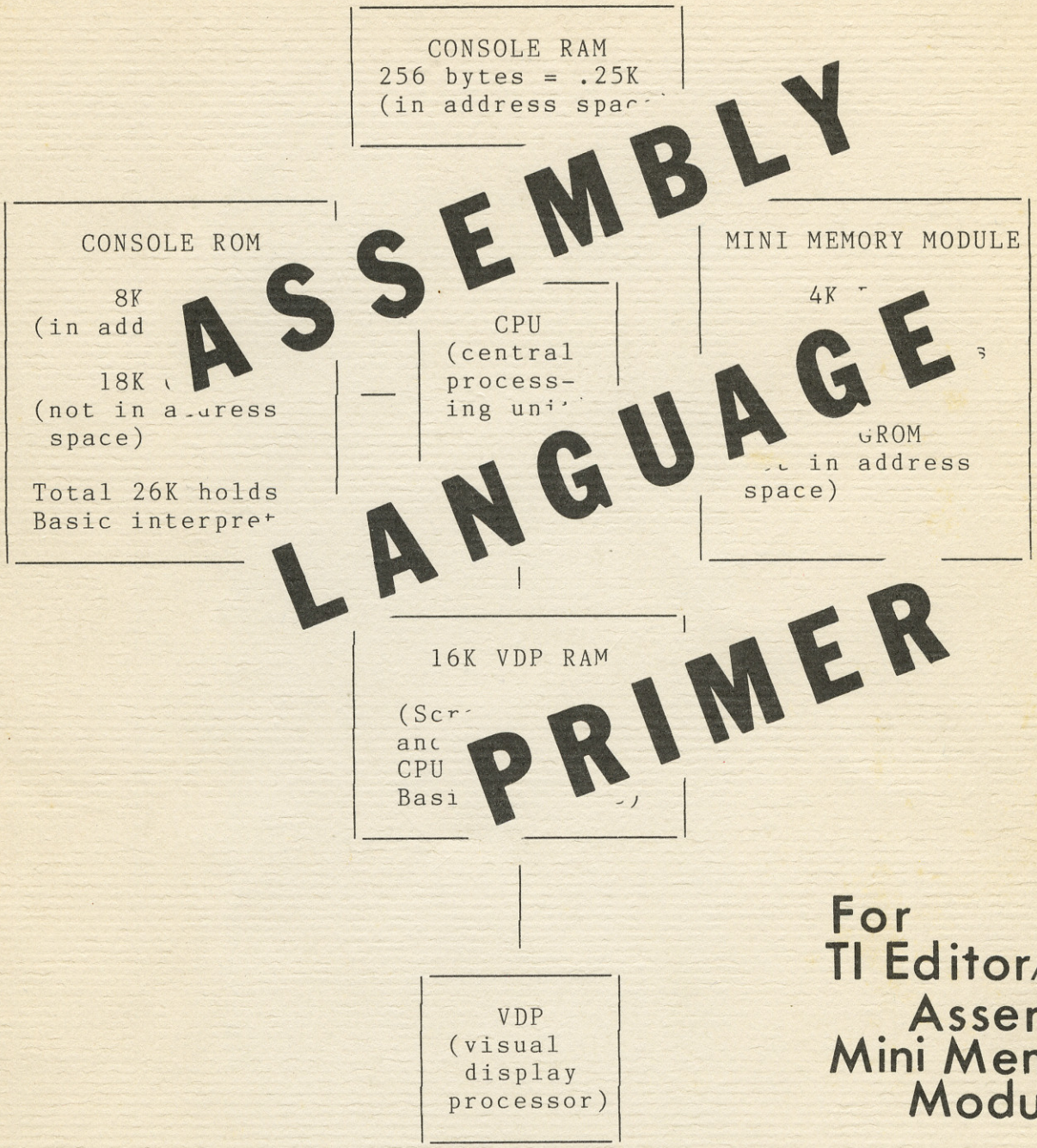
TI HOME COMPUTER

ASSEMBLY LANGUAGE PRIMER



For TI Editor/ Assembler & Mini Memory Module

John T. Dow
with
Donna Borland Dow

TI Home Computer

ASSEMBLY LANGUAGE PRIMER

An introduction to assembly language programming for Basic programmers.

by
John T. Dow
with
Donna Borland Dow

COPYRIGHT 1984 by John T. Dow
Pittsburgh, Pennsylvania 15217

All rights reserved. No part of this book may be reproduced in any form or by any means without permission in writing from John T. Dow

Acknowledgement

Several people helped by reviewing and criticizing prepublication versions of the primer. Special mention must be made for the assistance of Roy Caldwell, Walt Dollard, Henry Jaroszynski, and Dean Striegel

Table of Contents

Preface

Programming can be fun. Many owners of the TI-99/4A have learned Basic - some have taught themselves - and they have spent many sometimes frustrating but mostly rewarding hours designing, writing, and using their own programs.

This book is for those who want to progress from Basic to assembly language programming. The challenge is much greater, but the added enjoyment makes it worth the effort. In addition to the thrill of solving more difficult problems, your pleasure will be increased because you can do much more with the computer, and programs will run about 200 times faster than in Basic.

The book was written for those who know nothing about computers other than what they have learned with the Home Computer and Basic. It will help you learn about the TI computer and its assembly language. It will also improve your understanding of Basic on the TI and in fact will teach you many things about computers in general.

This book was written to be read from the front to the back. It is not a reference manual that can be read in any order. The aim of the book is to present facts mostly by means of examples rather than by formal definitions and statements. Sometimes this means that a particular presentation may not cover every detail, but they will be covered later when you should be better able to understand them. Try the test programs to understand the various instructions of the machine. If you do not understand something when you first read it, do not spend an inordinate amount of time on a particular detail but continue past it. You may understand it from a later example or if you return to it later.

This book was written based on the specifications in the TI Editor/Assembler Manual and without access to any "inside" information about the TI 99/4A Home Computer.

Needed Equipment

You do not need to have a computer equipped for assembly language programming to read and learn something from this book. However, since people learn better by doing than by reading, you should equip your computer properly. There are several **system configurations** that will work.

The best system is clearly TI's Editor/Assembler: however, to use it, you need the disk storage system and memory expansion. All of this equipment is much more expensive than the computer itself, so many people elect to work with the Mini Memory Module instead. Whichever way you do it, you definitely need the manual that comes with the TI Editor/Assembler since that is the reference manual for the computer.

If you buy the Mini Memory Module, you will receive the Line-by-Line Assembler. This has some severe limitations, such as not allowing you to store the source for your programs on cassette (as you can with Basic programs), and its editor does not allow you to insert or delete statements. You would probably not care to use it except perhaps for some simple experiments.

The Dow Editor/Assembler will make the use of the Mini Memory Module much more enjoyable because it has full editing capabilities and allows you to put programs on tape as you do with Basic. It also allows the use of all of the Module's available memory. Furthermore, its usefulness increases if you add a printer, expansion memory, or disk storage to your system. If you begin to use TI's Editor/Assembler, you can list your Dow Editor/Assembler programs to disk files and then convert them to compatibility with TI's with a few simple editing steps.

If you do use the Dow Editor/Assembler on a minimal system but then want to build it up, my recommendation would be to add a printer as soon as possible because any serious programming effort (whether in Basic or assembly language) goes much better if you have an accurate, up-to-date listing.

This book is written to be used with either the Dow Editor/Assembler or the TI Editor/Assembler. The syntax differences between the two assemblers are minimal and explained in the Dow Editor/Assembler manual; this book will usually show both or show the TI syntax. If you use the TI Editor/Assembler, you presumably would also have Extended Basic, so some of the examples which mix assembly language with Basic will assume that you are using Extended Basic.

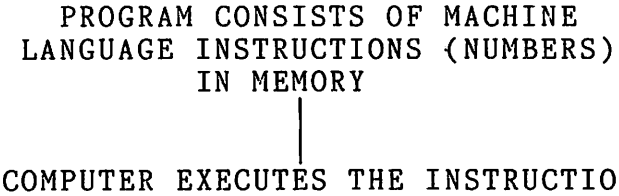
Needed Equipment

If you have the TI-Writer word processor, you might want to use its editor rather than the TI Editor/Assembler's editor because the files are compatible and the TI-Writer editor is much nicer. Additionally, if you use the TI-Writer editor for word processing, there is no need for you to learn to use two editors. If you do use it, set the tabs at 7, 12, and 25, and set the right margin at 58 (to prevent remarks from extending past the right margin when you select the list option during assembly).

A Brief History of Programming Languages

This history is only for illustrative purposes and is therefore somewhat oversimplified.

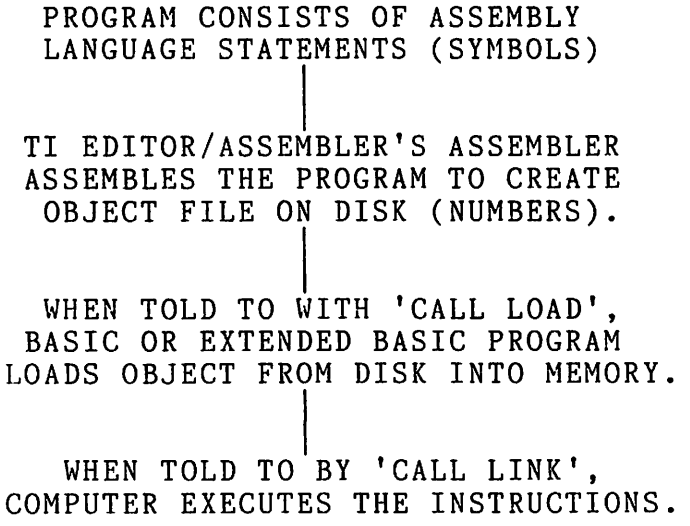
When computers were first built, there were only **machine languages.** These languages were very primitive and strictly oriented to the computer, for which there was no supporting software. The first programmers were true pioneers because programs consisted of a series of numbers, and to write a program in such a language, you had to know what all the numbers meant. (There were other complications as well.) In a nutshell, machine languages are great for machines but terrible for people. This simple diagram shows the relationship between a machine language program and the computer itself.



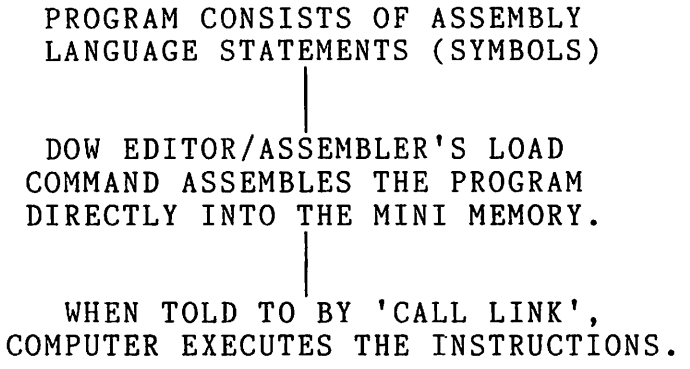
The machine language for one computer usually bears no resemblance to that for another. This makes it difficult to learn to program for more than one computer.

The next step up from machine language was **assembly language.** This is another generic term, meant to apply to languages in which the human programmer does not have to remember the actual numbers used by the computer to represent its instructions. Thus, these languages are easier to use because we humans do better with words (even nonsensical ones) than we do with numbers. Programs consist of statements with symbols and labels. Symbols and labels are like names in Basic; they are made of letters and, if you wish, numbers. Examples are: MAX, V5, BUF, and TOP. Statements in assembly language programs are turned into the numbers of machine language one statement at a time. This process is called **assembling.** It is performed by the assembler program included with the TI Editor/Assembler or by the LOAD command in the Dow Editor/Assembler. The diagrams on the next page show how assembly language statements, the assembler, and the computer are related.

This diagram shows how the TI Editor/Assembler is used to assemble, load, and execute an assembly language program.



This diagram shows how the Dow Editor/Assembler is used to assemble and execute an assembly language program.



Note that, like machine languages, assembly languages are unique to their respective computers.

As we shall see time and again throughout this book, assembly language still requires the programmer to know the particular computer in great detail. This degree of detail makes mistakes easy to make, and the programming effort required detracts from the problem you wish to solve when writing the program.

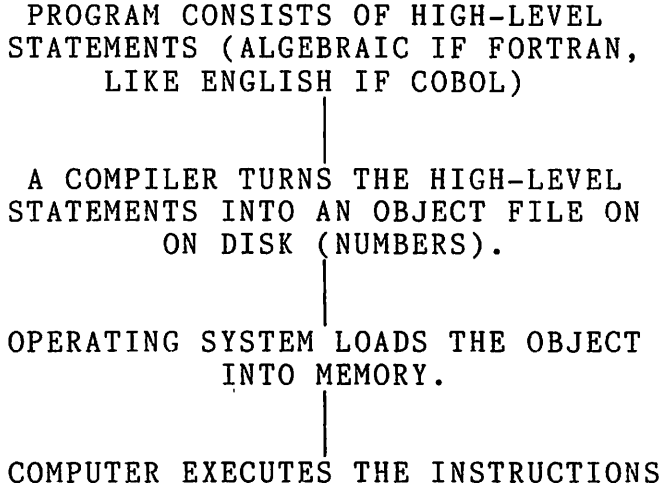
Next after assembly languages came several **procedural languages**, first Fortran, and then many others, including Cobol. These use symbols just as assembly languages do, but they are more advanced languages. Statements do not correspond to individual machine language instructions but may look like algebraic equations (as in Fortran) or like English sentences (as in Cobol). Because each statement generates several or even many machine language instructions, these are much easier to use than

assembly languages.

Fortran is still used, mostly in engineering or scientific applications. Cobol is still used widely in commercial data processing. Other somewhat similar languages include ALGOL, PL/I, and Pascal.

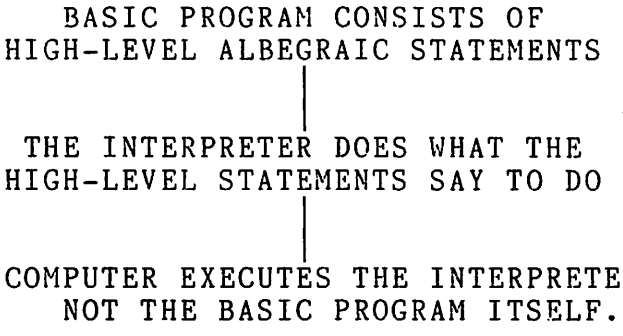
These are almost always **compiled languages** - this means that a program, called a **compiler**, translates **high-level** algebraic or English statements into a series of corresponding machine language statements. The programmer no longer has to understand the details of the machine, and indeed the same program with little or no changes may be compiled on very different computers. The advent of compiled languages greatly enhanced the productivity of the programmers.

Here is a diagram to show how high-level compiled languages work. (The languages used as examples, Fortran and Cobol, are not available on the TI Home Computer.)



The last class of languages I will discuss includes Basic. The programmer does not have to understand very much about the computer to use Basic, and indeed the TI-99/4A is especially good on this **user friendliness** issue. In this way, a language like Basic is similar to a language like Fortran. However, unlike compiled languages, a language like Basic is not translated into machine language to be executed by the computer. Instead, a program called an **interpreter** resides in the computer and does what the program says to do. You can think of the Basic program as the data for the interpreter program. In the TI, the Basic interpreter is written in machine language and resides, permanently, in the console. The simple diagram on the next page represents how Basic is interpreted.

A Brief History of Programming Languages



Compiled languages and interpreted languages are both high-level languages. That is, the programmer does not have to understand the intricacies of a computer to write a program to run on it. One language may in fact exist in both forms: for instance, Pascal and Basic exist in both forms, although not necessarily on the same computer. The chief differences between the two types are that 1) compiled programs run faster and 2) an interpreter can easily offer the programmer aids (such as BREAK and TRACE) to make programming easier.

It is important for the novice assembly language programmer to realize the vast difference between a high-level language such as Basic and a **low-level** language such as assembly language. To program in Basic, you had to learn to use its vocabulary and its constructs (such as GO TO, FOR...NEXT, and numbers versus strings), but much of what you learned had to do with organizing a solution to your problems in those terms. With assembly language, much of what you learned in Basic will still be useful, but you must learn a whole new set of concepts and much more detail.

Bits, Nybbles, Bytes, Words, and Hex

Terms such as "bits", "nybbles", "bytes", "words", and "hex" may sound somewhat like bits of a conversation at a witches' convention. However, they are several of the terms which stand for some concepts which are necessary for assembly language programmers to understand. They all have to do with the way we talk about data that is stored in computers.

Let us start with **bytes**. You could think of bytes as being characters, including the characters on the pages of this book. When I wrote this page with TI-Writer, TI's word processor for the TI-99/4A Home Computer, each character was stored in the computer's memory as I typed it in. That is, each character is stored in a byte of memory. Each byte sent to the printer specified which character was to be printed.

However, not only characters are stored as bytes in the computer's memory. The smallest directly addressable unit of memory is a single byte, so in fact, <u>everything</u> is stored in bytes.

Each byte in memory has its own name tag, called its **address**. As a member of a "computerized society", you should not be surprised that the address is a number. (Addresses are discusssed in greater depth in the chapter of this book that explains the memory of the computer.) For now, the important point is that each address refers to a unique byte in memory.

Sometimes memory is considered to consist of **words** rather than bytes. On the TI, a word is simply two bytes. However, the two bytes must occupy exactly adjacent locations, and the first byte must have an even address. For example, the bytes at locations 0000 and 0001 are one word.

It is important to remember that word addresses must be even. If you forget and use an odd address (or compute an odd address), the computer will not indicate an error condition but will simply use the next lower even address. For example, if you used 0001 for a word address, it would be treated as 0000. If you inadvertently make this mistake, it can be very difficult to track down. One of the skills of an assembly language programmer is therefore a constant alertness for odd addresses when only even will work properly.

Suppose the computer retrieves a byte out of its memory: just what does one look like? To understand them in better detail, it is necessary to move down to the next level of specificity, bits. Although I said earlier that you could think of bytes as characters, you also need to learn to think of bytes as groups of eight bits. A **bit** is a "BInary digiT." Each bit can be either 0 or 1. A byte is therefore eight 0's, 1's, or a

combination of both.

If you haven't tried using the Basic subroutines CHAR and HCHAR, try them now. This will help you understand just what bits and bytes are. Start by looking at this matrix, which defines a pattern.

```
10000000
01000000
00100000
00010000
00001000
00000100
00000010
00000001
```

As you know, you can redefine characters on the TI to have any shape you want them to have. Each character is eight bits across and eight bits down. (When talking about the screen display, the term **pixels** is essentially synonymous with bits.) The pattern of 0's and 1's above can be made to represent a shape which is a line sloping to the left. Each 0 represents background color, and each 1 represents black. (These colors can be changed by calling COLOR.) This is how you specify the pattern above with a call to the CHAR subroutine:

```
CALL CHAR(100,"8040201008040201")
```

To understand how the string "8040201008040201" represents the matrix of 0's and 1's, we need to appreciate how much more compact the string is than the matrix, and just how important this is. The matrix has 64 0's and 1's, while the string only has 16 characters. This compactness is important because it shows a big difference between the way people and computers work with symbols. Computers work well because they are as simple as possible. Computers are also very fast, so they make up for doing simple things by doing many of them in rapid succession. We humans, on the other hand, are very slow by comparison. We make up for being slow by doing a lot at once.

Let me make the point clearly. Look at these three character strings briefly, one at a time, and turn away from the book and try to repeat each in turn.

```
1)   0111110110001100
2)         7D8C
3)        32140
```

If you are like everyone else, the second string was easier to remember than the first, and the last was easier than the second. In fact, the first string was much harder to remember, if in fact you remembered it at all. You may be surprised to real-

ize that all are numbers and have the same value. The first number is in the **binary** number system, the second is in the **hexadecimal** number system, and the third is in the **decimal** number system (which is what we humans normally use). The first is what computers "prefer", while we humans do much better with the second and third.

The first is difficult to remember because it is so long, even though the symbols are familiar and simple. The second is easier because it is shorter, but is uses a strange mixture of characters and numbers so it is not as easy as the third.

The word "hexadecimal" comes from the Greek word for sixteen. The list below shows you the correspondence between the decimal, hexadecimal (or "hex" for short) symbols, and binary values. There are exactly 16 possible combinations of four 0's and 1's. They are listed here in numeric order, representing the values 0 through 15. Note that decimal and hexadecimal are the same for 0 through 9 - hexadecimal must represent the values 10 through 15 with a single character, so the characters A through F are used as if they were digits.

CORRESPONDENCE BETWEEN THREE NUMBER SYSTEMS

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Look at the first two versions of the number again, as shown below. This time, the binary value has been split into groups of four digits, with the hex digit just below it. Also, the decimal equivalent for each group is shown. (Caution: Grouping these digits together does not give the same as the decimal equivalent for the entire number.) You should be able to see how each binary group exactly matches the corresponding hexadecimal group.

Bits, Nybbles, Bytes, Words, and Hex

Clearly, hex is a nice shorthand for cumbersome binary strings. The advantage of hex over decimal can be seen here - in all cases, hex only requires one digit, whereas in some cases decimal requires two digits.

1) binary	0111	1101	1000	1100
2) hex	7	D	8	C
3) decimal	7	13	8	12

Let's go back to the preceding matrix. As displayed below, the matrix has been modified so that it consists of two sets of four columns. To the right of the matrix of 0's and 1's are two columns of numbers, with the values 0, 1, 2, 4, and 8. The second two columns are simply hex, while the 0's and 1's are binary. Although you can actually see the pattern better when represented in binary, the hex method of coding is preferable because it has only one fourth as many digits.

```
1000 0000  80
0100 0000  40
0010 0000  20
0001 0000  10
0000 1000  08
0000 0100  04
0000 0010  02
0000 0001  01
```

Because hex is more convenient for humans to use, the pattern is defined in the subroutine call as character 100 by

8040201008040201

rather than by

1000000010000000010000000000001000000001000000001000000001

To actually see what this character looks like, and thus learn to visualize bits (or pixels), try this simple Basic program.

```
100 CALL CHAR(100,"8040201008040201")
110 CALL CLEAR
120 CALL HCHAR(10,10,100)
```

You have now seen that eight bits can be represented by two hex digits. Eight bits are also a byte, so sometimes each of the two hex digits in a byte is called a **nybble**. A nybble is thus four bits.

Bits, Nybbles, Bytes, Words, and Hex

As stated above, computers do well with simple things. Let's briefly look at the binary number system to see why virtually all computers use it rather than the decimal number system which we humans prefer. Here are the complete addition and multiplication tables for binary.

ADDITION

```
  0      0      1      1
+ 0    + 1    + 0    + 1
  0      1      1     10
```

MULTIPLICATION

```
  0      0      1      1
x 0    x 1    x 0    x 1
  0      0      0      1
```

Computers are built to use binary arithmetic because it is so simple. In fact, being binary also simplifies the memory and data transmission.

It is not really necessary for you to be fluent in binary or hexadecimal arithmetic. You do need to understand them at a fundamental level and be able to occasionally do an addition on hex numbers. What is more important is that you learn to think of bit strings and their equivalent hex strings, since not only numbers but also characters and the machine's instructions are bit strings.

Having just said that fluency in hex arithmetic is not necessary, let us look at a few examples.

HEXADECIMAL	DECIMAL
0005 + 0004 = 0009	5 + 4 = 9
0005 + 0005 = 000A	5 + 5 = 10
000A + 0009 = 0013	10 + 9 = 19
7E48 + 006E = 7EB6	32328 + 110 = 32438

You need to understand another important fact about binary and hex, and that is how most computers handle negative numbers. We have been taught to write a number the same way, whether positive or negative, except that the minus sign "-" precedes a negative value. However, nearly all computers represent negative values by using the **two's complement** of the positive value. To get the 2's complement of a value in binary, start by changing each 0 to a 1, and change each 1 to a 0. (Making these changes to each of the bits is called **complementing** the bits. The result is called the **one's complement**.) Then add 1 to the result, forming the two's complement. In changing each bit, you

must remember to change all the leading 0's as well. Thus, because the TI is a 16 bit machine, you need to complement each of the 16 bits, then add 1 to the rightmost bit. This example shows how to complement the value 1:

```
      BINARY
0000000000000001 Before
1111111111111110 After complementing each bit
1111111111111111 After adding 1

  HEXADECIMAL
     0001        Before
     FFFE        After complementing each bit
     FFFF        After adding 1
```

This shows that -1 is 1111111111111111 in binary and FFFF in hex. Compare that to a very large, positive number, the value 32,767. It is 0111111111111111 in binary and 7FFF in hex. It only differs from -1 in that the left-most bit is a 0 instead of a 1.

The good thing about using two's complement notation for negative numbers is that they can be added with exactly the same hardware logic as a positive number, since negative numbers are so large that the addition may cause overflow, thereby resulting in a positive value again. For example, let's add 5 and -1. In hex, this is 0005 + FFFF. It is shown here with the carries above and in parentheses.

```
(1)  (1)  (1)  (1)
      0    0    0    5
      F    F    F    F
     ---------------
(1)   0    0    0    4
```

Adding the digits on the right side (5 + F) yields decimal 20, or hex 14. Carry the 1, and add 0 and F from the second to the right digits. That yields decimal 16, or hex 10. So carry the 1 again, and again. The final result is 10004. But since the 1 on the left is lost (because it is the 17th bit), the result is 0004, which is indeed the sum of 5 and -1.

Bits, Nybbles, Bytes, Words, and Hex

Here are several values, shown in hex and decimal. These are values you are apt to see frequently, so you should learn to recognize them.

HEXADECIMAL	DECIMAL
0001	1
000F	15
0010	16
00FF	255
0100	256
1000	4096
FFFF	-1
FFFE	-2
FF00	-256

Remember that if you work out the bits for a value such as FFFF, which would be 65,535, they may also be negative numbers in two's complement form.

After all this detailed discussion, a brief summary is in order. A **bit** (short for "binary digit") is a 0 or a 1. A group of four bits can be represented by a single hex digit, called a **nybble**. Two nybbles, or two hex digits, make eight bits, and thus two hex digits represent a byte. Eight bits therefore also make a **byte**. Two bytes make 16 bits, and if the address of the first is even, 16 bits make a **word**. This is shown here.

TERM	DEFINITION	RANGE OF VALUES (in hex)
Bit	Binary digit	0 and 1
Nybble	4 bits	0 through F
Byte	8 bits=2 nybbles	00 through FF
Word	16 bits=4 nybbles=2 bytes	0000 through FFFF

Finally, the computer does not care what is stored in a location. It could be a character, a numeric value, or an instruction. To the memory of the computer, a byte is just eight bits regardless of what it may mean to you and regardless of how your program is supposed to use it. In fact, you frequently cannot tell what is stored in a byte if that is all the information you have. For instance, hex 7041 could be the characters "pA", a *sub*tract instruction, the value 28737, part of the definition of a character, or many other things. Using the hexadecimal number system is merely a convenient way to write all these different things.

Appendix C contains a Basic program you can use to convert values from decimal to hexadecimal and back.

The CPU and Other Processors

All computers have one (or more) central processing unit, called the **CPU**. For microcomputers, the entire CPU is on one chip. In the TI Home Computer, it is the **TMS9900 chip**, which has been available since the 70's. When it was introduced, it was a departure from the norm because it performs arithmetic on 16 bit integers (more on this later in the book) rather than on 8 bit integers. Today there are many 16 bit machines, including the IBM PC and the TI PC, and many somewhat similar machines. However, these other 16 bit machines use a different processor and are not packaged as inexpensive home computers.

When you learn assembly language for the 99/4A, much of what you learn is the instruction set for the 9900 processor. However, there is much more to the TI Home Computer than the CPU. There are also the visual display processor (called the **VDP**), a processor for the cassette port, the sound generator, and the speech synthesizer (not included in the console itself).

If you plan to write programs to enhance the power of Basic, you will not need to learn as much about these other processors. However, if you want to write an arcade game program, you will not only have to learn how to program the CPU but also how to control these other processors. (I say "control" because you can tell them what to do but you don't really program them to the extent that you program the CPU.) Note that when you write in Basic, you do have some limited degree of control over these functions without having to understand them in much detail.

This book will emphasize programming the CPU since that skill is needed no matter what you do with assembly language. Once you have grasped the concepts of programming in assembly language, you should be able to advance into these other areas more readily by relying on the TI Editor/Assembler manual.

The Memory of the TI-99/4A

A very important part of any computer is the **high speed memory** that is accessed by the CPU. The memory is used to hold both program steps (i.e., instructions) and data. Computer memories can be given a new program step or data element to store as easily as the old ones can be retrieved: the ease with which a program can be changed and stored, when compared to the difficulty of changing something which is "hard-wired" or mechanical, is a major reason why computers have been a revolutionary new step in our post-industrial age.

A microcomputer's memory consists of **bytes.** This was discussed briefly in the chapter on nybbles, bytes, and so forth. You are also familiar with bytes indirectly at least from Basic - each character in a string is a byte.

The size of the memory is measured in K's. The letter "K" is used for the value 1024, which is 2 raised to the tenth power. This is a convenient value because it is nearly equal to 1000. The console RAM in the TI is said to be 16K, which is approximately 16,000. The exact value is 16 times 1024, or 16,384 bytes. Note that this terminology actually mixes two number bases - base 10 and base 2 - but does so in a way that lets us easily understand approximately how much memory is involved because we are familiar with the decimal value 1000.

Memory is accessed by addresses. Each byte has a unique address, which is a number 0 or greater. Each computer is limited in the number of addresses it can handle. For instance, the TI-99/4A uses 16 bits for memory addressing. With 16 bits one can count from 0 to 64K minus 1, or 65,535. This means that the **address space** for the TI is 64K, because it can refer directly to that many bytes.

Because addresses are represented as binary values, the maximum memory size is nearly always an even multiple of two. Thus, various computers for home use may have 2K, 4K, 8K, 16K, 32K, or 64K.

A computer need not actually have all the memory in its address space. Because of this, you can buy memory expansion "cards" or "modules" for some computers, including the 32K card for the TI. Sometimes part of the address space is reserved for special uses, and indeed it is on the TI, as we shall see later.

It is also possible to have memory which is not within the computer's address space. Most printers have their own memory, enabling them to accept data much faster than they can actually print it. This is called a "buffer". While this memory does not belong to the computer itself, it shows how additional memory can be used by the computer system without its being part of the

CPU's memory space. We will see that the 16K in the TI is in fact very similar to the memory in the printer.

Microcomputers have made extensive use of **ROM** - read only memory. This differs from traditional memory in that the CPU cannot store anything in it. The nice thing about ROM is that when you turn on the Home Computer, it can immediately come to life with its built-in Basic interpreter; without ROM, you would have to load Basic from an external memory, such as cassette or disk.

What is put in the ROM at the time of manufacturing can never be changed. The CPU does not know if some of its memory is ROM, so in fact the essential character of the computer has not been altered by the use of ROM. The TI-99/4A has 26K of ROM, quite a lot by current standards. Command modules (plugged in to the slot on the top right of the machine) also contain ROM; in the case of Extended Basic, there is an additional 36K of ROM.

One type of ROM, called **GROM**, was invented and patented by TI. GROM can only be used in certain ways, which will be described below.

Memory into which new data can be stored needs a name to distinguish it from ROM. In the old days, computers only had one kind of memory, called "core memory". You may still hear this term used, but due to different memory technologies now in use, the usual name today is **RAM**, which simply means "random access memory". This is really a very poor term to use, since other forms of memory, including ROM and disk storage, are also random access. All that random access means is that the data can be retrieved from the memory in any order by specifying the address. The alternative form of retrieval would be sequential. A tape memory is a good example of sequential storage.

Remember that the address space of the TI is 64K. In fact, it has only 256 bytes of RAM for use by the CPU. These 256 bytes represent less than one half of one percent (.5%) of the potential memory space. The 256 bytes of RAM is called the "**PAD**" and is the only memory in the console which can be altered by the CPU.

Of the memory in the console which is in the address space, nearly all of it is ROM. Because the PAD is so small, you cannot write assembly language programs without using either the Mini Memory Module or the 32K memory expansion card - there simply is no place to put the program to run it.

Perhaps you are now wondering about the 16K that you know is in the console. This is indeed RAM, but unlike the 256 bytes, it really does not belong to the CPU. In this sense, it is like the memory in a printer. That 16K belongs to the **VDP**, the visual

display processor. This memory holds the screen image and color table for Basic. When Extended Basic is in use, it also holds the data describing any sprites you have. Because Basic and Extended Basic use screen display modes which do not use very much of the 16K RAM, most of it is unused by the VDP to display the screen. Therefore, the bulk of it is actually used by the CPU as a storage device for the Basic or Extended Basic program itself.

Because the 16K is not in the CPU's address space, assembly language programs cannot be stored there and run. Furthermore, the CPU can only access the memory sequentially, although it can reposition within the memory quite rapidly. You can think of the 16K VDP memory as a very, very fast cassette drive for the CPU. Every time the CPU needs to access something in the VDP RAM, it must reposition, then read sequentially. Although this is much faster than a tape, it is also much slower than if the CPU could directly access this memory within its address space. This is also probably the major reason why Basic on the TI is slower than on some other home computers, especially when one would expect it to be faster because it has a 16 bit processor as opposed to an 8 bit processor.

By now you have probably realized that the CPU has access to several different types of memory. The console and the modules contain ROM which the CPU can directly access and from which it can execute instructions. There is also **GROM** in both the console and modules, which the CPU accesses sequentially; this means that the CPU cannot execute instructions in GROM. Nor can it execute instructions in the VDP RAM; it can only read and write sequentially.

To summarize, there are several types of memory: ROM, GROM, RAM, and VDP RAM. The CPU can directly access any location in ROM or RAM, and this memory exists within the address space of the CPU. GROM and VDP RAM are only accessed sequentially and are not within the address space of the CPU.

Refer now to the table below. It shows the address space of the CPU in simplified form.

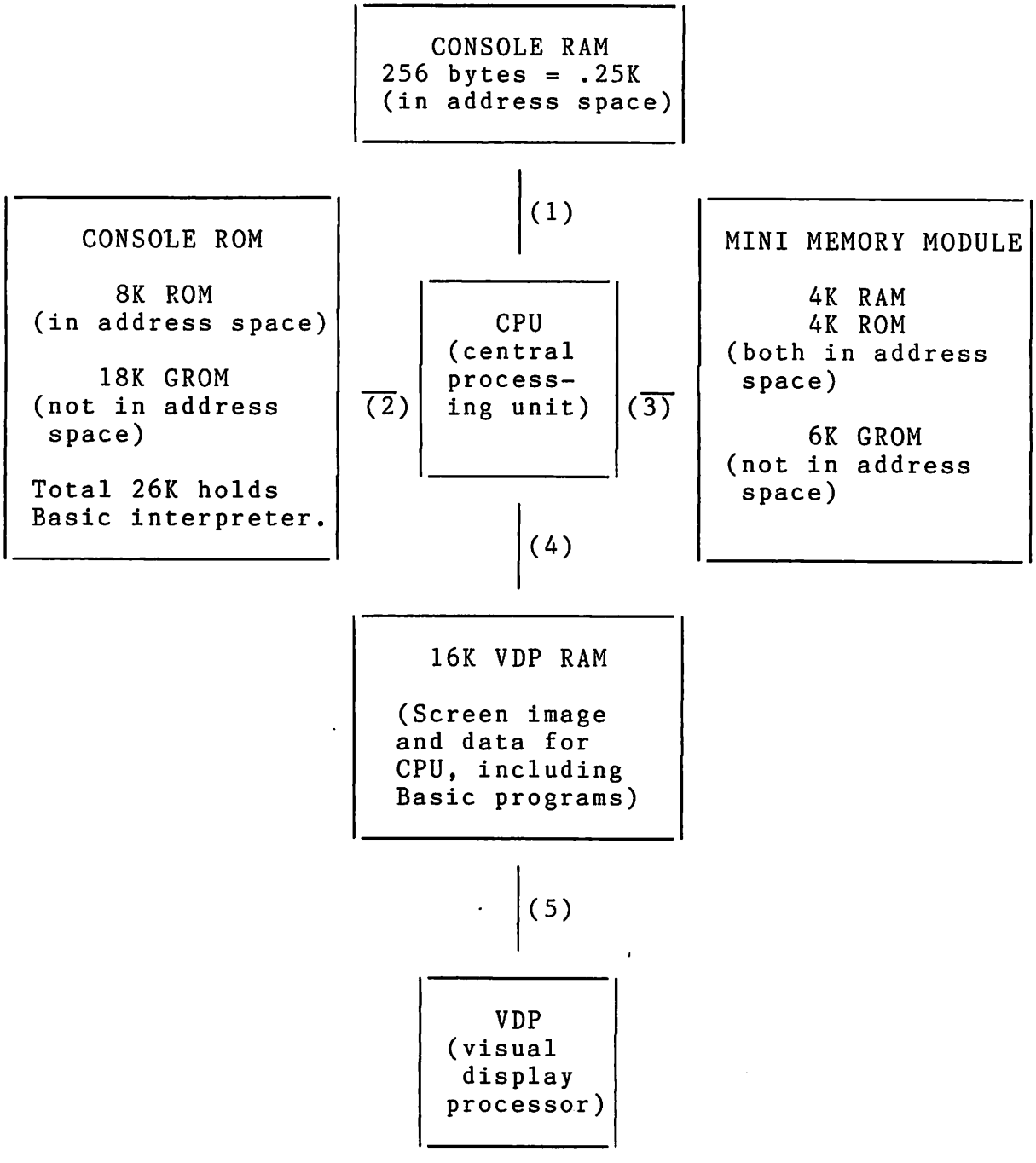
1st	8K	Console ROM
2nd	8K	Reserved for expansion memory RAM
3rd	8K	Reserved for peripherals
4th	8K	Reserved for modules (ROM or RAM)
5th	8K	Special uses, including 256 byte PAD RAM
last	24K	Reserved for expansion memory RAM

Notice that 32K, exactly half the 64K address space, is reserved for expansion memory RAM. Of the remaining 32K, 8K is held for modules, 8K is held for peripherals, and 8K has special uses. This leaves only 8K for ROM in the console. One of the

advantages of GROM is the fact that it adds storage without occupying the precious address space. Also, notice that the 16K VDP RAM is not a part of the address space of the CPU.

Perhaps a look at a picture will help clarify what has been said so far. The figure on the next page schematically illustrates the two processors and the several types of memories discussed so far. The command module shown is the Mini Memory Module; note that the typical module has no RAM, only ROM.

MEMORY STRUCTURE FOR THE TI-99/4A



(1) The CPU can read (retrieve) or write (store) directly from or to the 256 byte RAM.
(2) The CPU can read the 8K ROM directly and can read the 18K GROM sequentially.
(3) In the module, the CPU can read and write the 4K RAM, can read the 4K ROM, and can read sequentially the 6K GROM.
(4) The CPU can read or write sequentially the 16K VDP RAM.
(5) The VDP accesses its 16K directly.

The CPU and Registers

In order to understand the instructions for the CPU, it helps to know a few things about the CPU itself. That can be learned with a comparison to Basic.

In the TI, there are three special purpose registers contained within the CPU which the programmer needs to know about. TI calls them **hardware registers**. These are discussed below.

1) The PC register.
2) The status register.
3) The workspace pointer.

A typical statement in Basic is "J=J+K". This means to add the values of J and K together and to then put the sum back into J. The programmer only knows J and K by their names. He need not be much concerned about their values, since Basic supports a large range (10 to the 64th power) and great precision (13 to 14 digits).

In assembly language, this would be "A @K,@J". This also means to add the values of J and K and then to put the sum into J. Unlike Basic, the programmer must be sure to reserve memory locations for both J and K (we will see how to do this later). He must also be concerned about the range of values both J and K may have in his program, since the CPU only performs integer arithmetic. Since the TI is a 16 bit machine, values may range from -32,768 to +32,767. Integers, by definition, have no fractional part.

After a statement in Basic has been interpreted, the next statement in sequence is interpreted (unless the statement was a GOTO, IF, or FOR...NEXT loop). The interpreter has to keep track of which statement it has just interpreted and be able to determine which is next. In the same way, when executing an assembly language program (that has been assembled into machine language), the CPU executes statements one at a time, in sequence, unless there is a special instruction to alter the order. The CPU also has to keep track of which statement it has just executed: it does this with the Program Counter register, or **PC**.

The CPU uses the PC register to keep track of the current location of the program. The PC register is simply a 16 bit counter, able to count from 0 to 65,535. This is the size of the CPU's address space. Thus, the value in the PC is the address of the next instruction to be executed. (Note: it is not the value of the instruction last executed.) Many instructions are two bytes long (some are four or six); the PC is therefore automatically incremented by two (or four or six) after executing these instructions. In this way it always points to the next

instruction.

If the instruction is a jump (the equivalent to a Basic GOTO), the instruction does not actually make anything "jump" - instead, it simply puts a new value into the PC register.

Whether the PC is automatically incremented or is loaded with a new value, when the CPU is ready for another instruction, it uses the value in the PC register to get two bytes from ROM or RAM, starting at the location pointed to by the PC. Those two bytes form the next instruction. Once loaded into the CPU, into a special instruction decoding register, the CPU proceeds to execute that next instruction.

The PC is one of several registers in the CPU. Some registers are 16 bit counters. Other registers hold values which are not necessarily addresses, an example being the instruction decoding register mentioned above.

The status register (**ST**) is a means by which the program (software) can communicate with the CPU (hardware) about miscellaneous conditions that may arise. Perhaps you have used a variable in a Basic program which was set as the result of a logical test. For example, if you have a variable in your program called VALUE, you might test it like this:

```
TOOMUCH = VALUE > 10000.
```

Having done this, your program can easily test for the condition TOOMUCH any number of times. The variable TOOMUCH has the value FALSE or the value TRUE. This type of variable is a **flag.** You can think of the status register as a set of such flags, each bit in the register being a flag. The exact meaning of the various bits in the status register will be discussed later.

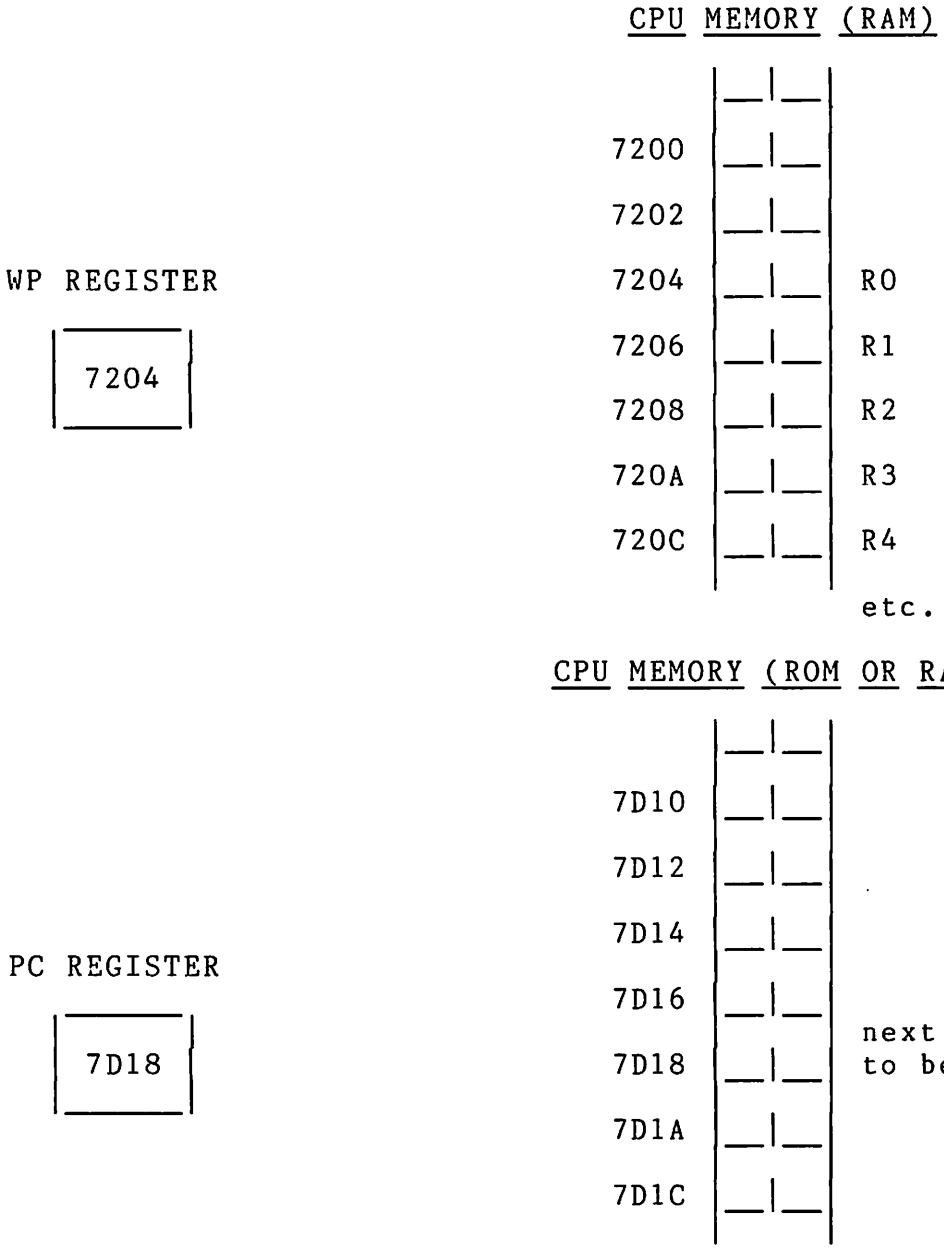
The two registers just discussed are necessary but hardly sufficient for a computer. Other registers are needed for general purpose tasks, such as arithmetic operations and comparisons. The TI has a set of 16 general purpose registers. These registers are different than the PC and ST registers because they are actually memory locations.

The workspace pointer register (**WP**) is a special purpose register which points into memory to the 9900's general purpose registers. TI calls these **software registers.** The WP must point to a 32 byte region of RAM which the programmer has designated to be the 16 general purpose registers. It is the responsibility of the programmer to make sure the WP contains the address of these 32 bytes. (However, if you call an assembly language routine from Basic, the WP has been already set up to point to a 32 byte space and you need not worry about it.)

The CPU and Registers

These 32 bytes are taken two at a time to be 16 registers, numbered 0 to 15. Each has 16 bits. These registers are used to hold any 16 bit pattern the programmer chooses, including addresses (pointers), integer values (just as the variables J and K), characters, or instructions. In many cases they are used to perform arithmetic operations, instead of using variables. For instance, this is another addition instruction: "A R6,R9". It says to add the values of registers 6 and 9, then to put the sum into register 9.

This figure will help show what the PC and WP registers do.



The CPU and Registers

The WP and PC registers are represented by boxes on the left of this diagram, while memory is represented on the right. Since the registers point to even addresses, memory is shown as two columns - each column is a byte, and the two bytes together make a word. The addresses shown next to the memory are for the left column. All numbers are hexadecimal.

The WP register indicates that the address of the software registers starts at 7204. That is, register 0 is locations 7204 and 7205, register 1 is locations 7206 and 7207, and so forth. The PC shows that the next instruction to be executed is at locations 7D18 and 7D19.

Note that the PC can point into ROM, but since the WP points to registers which are used for computations, it would not be useful for the WP to point into ROM.

Very often the WP points into the PAD RAM area in the console. If there is no additional memory on the computer (Mini Memory Module or Memory Expansion), the WP has to point into the PAD (locations >8300 through >83FF). Because the PAD is faster than other memory, it often makes the most sense to use the PAD for the register space.

The First Program

In this chapter you will enter and run the simplest possible program. The purpose will be to learn how to go through the mechanics of getting a program assembled and loaded as well as to begin to learn some detail of the computer's instruction set.

The simple program does nothing except return to the Basic calling program. It is therefore similar in concept to the Basic statement "RETURN". Frequently what we program in assembly language is a subroutine, either called from Basic or from another assembly language program. Therefore, the return statement is used quite often.

The essence of a return statement in any language is simply to return control to the point from which it was transferred to the subroutine. In Basic, the interpreter performs this function by storing the return on a stack when the subroutine is called with GOSUB.

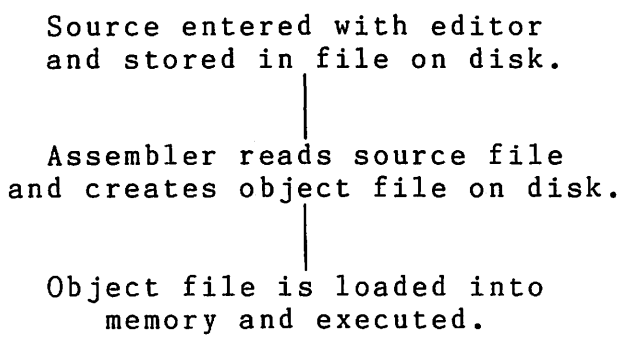
A subroutine call in assembly language consists of executing the "branch and link" instruction, written "BL". This sends control to the address (designated in the instruction) which is the entry point of the subroutine. So that return is possible, it also stores the return address in register 11. Thus, to return from a subroutine, the programmer must make sure to transfer control back to the address stored in register 11. A "branch indirect" will do this: it is written "B *R11".

Since this instruction is used so often, the TI Editor/Assembler allows you to abbreviate it as simply "RT", as though it were a special instruction. However, you can still use "B *R11" and have exactly the same result.

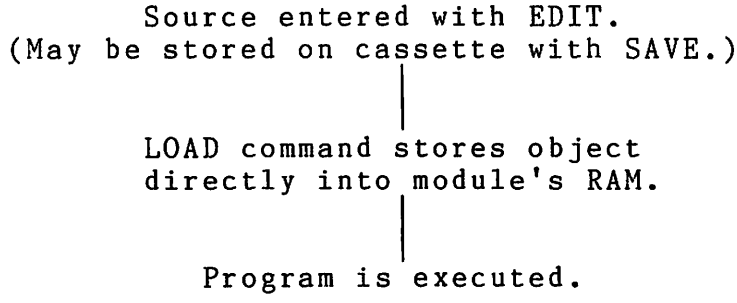
There are two terms that are used for assembly language but are not needed for Basic programming. First, **source code** refers to a program as you write it, whether with the TI or Dow Editor/Assembler. The source program has labels and names. Second, **object code** refers to the program after assembly. It no longer has labels and names.

With the TI Editor/Assembler, you key in the source using the editor: you must then store it in a disk file. This file in turn is read by the assembler, which generates the object file, also on disk. The object file is loaded into memory by CALL LOAD from Basic or Extended Basic.

Here is a diagram to show this sequence.



With the Dow Editor/Assembler, you key in the source using the EDIT command. Using the SAVE command, you can store it on tape if you wish. The LOAD command generates the object code and puts it directly into the memory. This is how it is done.



Let us now follow the steps for entering, loading, and running this program with the TI Editor/Assembler. (The instructions will then be repeated for the DOW Editor/Assembler.)

Step1) Insert the Editor/Assembler Command Module and the diskette labelled "Editor/Assembler Part A." (Actually, you should make a copy of TI's diskette and use the copy. If you delete the DEBUG files from your copy, there will be room on the same diskette to store some of your own programs and you will not have to switch diskettes repeatedly.)

Step 2) Press any key, then press 2 to select the Editor/Assembler.

Step 3) Press 1 to select the editor.

Step 4) Press 2 to edit. (Wait while the editor loads from disk.)

Step 5) Enter the program. It should be displayed on the screen like this.

```
        DEF   TEMP
TEMP    RT
        END
```

These instructions give the actual key strokes needed to enter the program. Note that TAB means FCTN 7.

First line ...
- FCTN 8 (to insert a line)
- TAB
- type DEF
- TAB
- type TEMP
- ENTER

Second line ...
- Type TEMP
- TAB
- type RT
- ENTER

Third line ...
- TAB
- type END
- ENTER

Press FCTN 9 twice to leave the editor.

Step 6) Press 3 to save the program. Reply Y to the prompt about variable 80 format. For a filename, you could specify "DSK1.TEMP". Switch diskettes if there is not enough space on the diskette which has the editor program. Press ENTER and wait while it writes the program to disk. Then press FCTN 9.

Step 7) Press 2 to assemble. If necessary, change back to the "Editor/Assembler Part A" diskette. Then reply Y to the prompt about loading the assembler. Wait while the assembler is loaded from disk. If necessary, replace the diskette containing your source file. When asked for the name of the source file, reply with the same file name given in step 6 above (DSK1.TEMP). For the name of the object file, you could enter "DSK1.TEMPOBJ". If you have a printer, you would give its name in response to the prompt for the list file name, otherwise just press ENTER. (Examples: "PIO", "RS232", "TP", "RS232.BA=9600".) For options, enter RLS if you have a printer and want a listing to be printed, otherwise just enter R. Now wait while it is assembled. You should get a message that there were no errors. Press ENTER.

Step 8) Press FCTN 9 to return to the master title screen.

Step 9) Press any key, then 1 for Basic. Enter and run this program with the Editor/Assembler module still inserted.

```
100 CALL INIT
```

```
110 CALL LOAD("DSK1.TEMPOBJ")
120 CALL LINK("TEMP")
```

The call to INIT prepares the computer to accept your program. For example, it copies some utility programs from ROM in the cartridge into the Memory Expansion RAM. The call to LOAD actually reads the object file from disk and loads it into Memory Expansion. The call to LINK transfers control from the Basic program to the assembly language program.

The file name specified in statement 110 must be the same as the name given for the object file during the assembly phase in step 7. The name passed to LINK in statement 120 must be the same as the name appearing on the DEF statement in the program in step 5. When you run the program, there will be a pause as the object file is read from disk and loaded into the expansion RAM by CALL LOAD, but the actual execution will only require microseconds.

Step 10) To prove that your program is actually being loaded, change the file name in statement 110 and you will get the message I/O ERROR 02 IN 110. To prove that is actually being called, change TEMP to XXXX in 120 and you will get the message PROGRAM NOT FOUND IN 120.

If you were to list this program during the assembly in step 7, it would look like this:

```
 99/4 ASSEMBLER
VERSION 1.2                                          PAGE 0001
  0001                    DEF  TEMP
  0002 0000 045B  TEMP    RT
  0003                    END



 99/4 ASSEMBLER
VERSION 1.2                                          PAGE 0002
    R0    0000      R1    0001      R10   000A      R11   000B
    R12   000C      R13   000D      R14   000E      R15   000F
    R2    0002      R3    0003      R4    0004      R5    0005
    R6    0006      R7    0007      R8    0008      R9    0009
  D TEMP  0000
  0000 ERRORS
```

On the first page, the numbers 0001, 0002, and 0003 in the column on the left are merely statement numbers. The 0000 on line 0002 is the **relative address** at which the statement will be loaded. Notice that the DEF and END statements have no address: that is because they generate no code to be loaded. The 045B is the code generated by the RT statement. It means B *R11 (branch indirect of register 11). The term "relative address"

means that the address shown is not the actual address at which the instruction or data will be loaded, but represents a relative amount to be added to the actual address when the program is prepared for loading into memory.

On the second page all the symbols and their values are listed. R0 through R15 were automatically defined by the R in the option specification in step 7. The only label explicitly defined was TEMP. The D by it means that it also occurs in a DEF statement.

Here is what the screen should look like after entering the same program using the DOW Editor/Assembler. (The value 7FE8 in the fourth line may not appear when you do this. The value displayed will be simply whatever happens to be at locations 701E and 701F when the program is entered.)

```
DOW EDITOR/ASSEMBLER
(C) JOHN DOW 1982
----->MINI 701E
>701E >7FE8  ?>7FF8
>701E >7FF8  ?.
----->EDIT
E->E
LOC LBL:OPCD OPERAND(S)
000     TEXT /TEMP  /
006     DATA >7118
008
006     DATA >7118
E->.
---->LOAD 7FF8
ADDR = >7FF8  OK?Y
....
NEXT = >8000
---->NEW
---->EDIT
E->E
LOC LBL:OPCD OPERAND(S)
000     B    *R11
002
000     B    *R11
E->.
---->LOAD 7118
ADDR = >7118.  OK?Y
.
NEXT = >711A
---->LINK TEMP
---->
```

This is a step by step explanation of what you do to enter, assemble, load, and run the program.

The First Program

Step 1) Put the Mini Memory Module into the slot on the computer. (Remember the admonition to turn off the computer while doing this so that you do not inadvertently destroy any data or program you may have already stored in the battery powered RAM in the module.)

Step 2) Press any key, then press 1 for Basic.

Step 3) If you want to clear out anything stored so far in the Mini Memory Module, type "CALL INIT" and press ENTER.

Step 4) Load the Dow Editor/assembler and run it.

Step 5) Next you have to make an entry in the **REF/DEF** table in the Mini Memory Module. This is a table of names and their associated addresses. It forms the link between the calling program, which uses the name, and the assembly language routine, which is loaded at an address determined by you. Since you also determine the name, you must enter both name and address into the table. Look at the diagram at the end of the chapter to see how **LFAM** (last free address in the module) points to the table, and the table points to the program's entry point. To make the entry in the table, follow the instructions in Section 7 of the Dow Editor/Assembler manual, except change 7500 to 7118. This is what you would enter:

Type MINI 701E, press ENTER.
Type >7FF8, press ENTER.
Type a period, press ENTER.
Type EDIT, press ENTER.
Type E, press ENTER.
Space 4 times, then type TEXT, space, type /TEMP /
(with two spaces after TEMP), press ENTER.
Space 4 times, then type DATA, space, >7118, press ENTER.
Press ENTER again.
Type a period, press ENTER.
Type LOAD 7FF8, press ENTER.
Finally type Y, press ENTER.

You will see four dots displayed as the four words are loaded into the Mini Memory Module's RAM. It will then say that the next location (that is, the one just after the last location loaded) is >8000.

Step 6) Key in the program. First, you will have to clear out the TEXT and DATA statements entered above.

Type NEW, press ENTER.
Type EDIT, press ENTER.
Type E, press ENTER.

The First Program

You are now ready to key in your program as shown below. It only has one statement. The spaces are necessary for the instruction (the "B") and the operand (the "*R11") to be in the correct columns. After you press return, the editor checks the statement for correct syntax. It then prompts for another statement.

> Space 4 times, type B, space 4 times, type *R11, then press ENTER.
> Press ENTER (to leave the editor's enter mode).
> Type a period (.), press ENTER (to leave the editor).

Step 7) Load the program into the Mini Memory Module. Type LOAD 7118 and press return. Notice that the value you enter here absolutely must be the same as the value you loaded into the table a few minutes ago. Furthermore, it must be the **entry point** of the program. (The entry point is the first instruction to be executed. Sometimes you might want other instructions or data to come first in a program, but in our simple example, the entry point is the one and only instruction.) The loader will ask you to confirm the value you entered before it proceeds with the loading: type Y and press ENTER

Step 8) You can now call the program. Type LINK TEMP and press ENTER. Since the program doesn't do anything, the Dow Editor/Ass- embler will just ask you for another command right away. However, if there was a problem with one of the steps above, you may get an error message. For instance, if the REF/DEF table was not loaded properly or the name entered with the LINK command (TEMP in this case) is not in the table, you will get a message such as "PROGRAM NOT FOUND IN 2810". In that case, you have to run the Dow Editor/Assembler again, which will wipe out your assembly language source program. Another possibility is that you may even lose control of the computer and have to turn if off and back on again to regain control!

Step 9) You can prove that you can call the program from any Basic program. Tell the Dow Editor/Assembler STOP, then enter the Basic command NEW to prepare to enter your own Basic program. This is all you need:

```
100 CALL LINK("TEMP")
```

When you RUN this program, it will terminate immediately. To prove that it actually called the program, change TEMP to XXXX and call LINK again: you will get the message "PROGRAM NOT FOUND IN 100".

You can type MINI 7118 to see the instruction that was assembled into the Mini Memory Module's RAM. It will display the value >045B, which is the machine language version of B *R11.

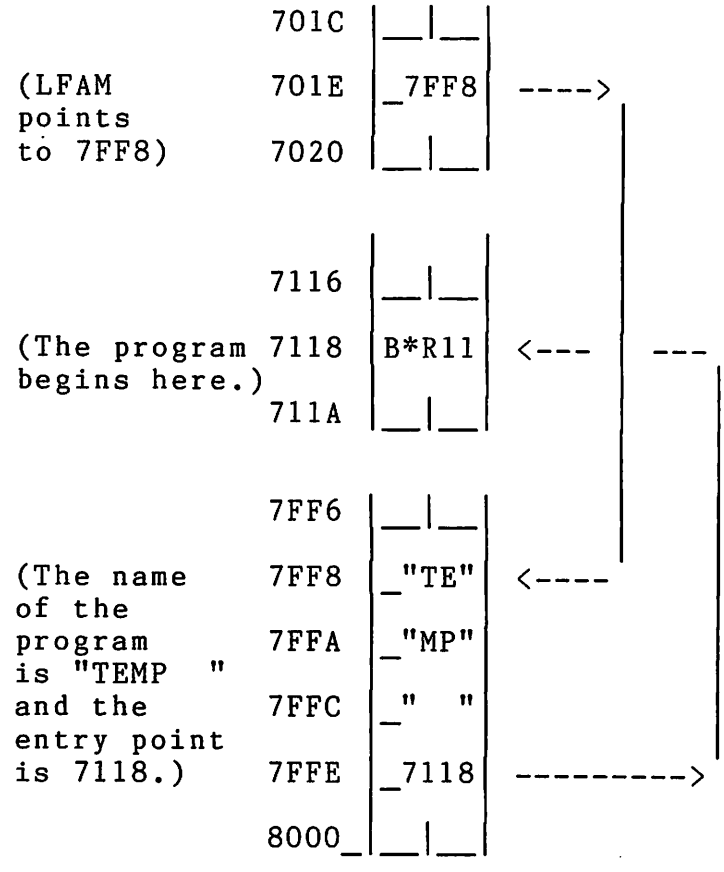
The First Program

If you have a printer and tell the Dow Editor/Assembler to list this program, it will display the prompt "OLD:" and "OK?". This shows you that there is no "old" label: you may enter a label by pressing ENTER and then entering the label in response to the prompt "NEW:". For instance, a label might be as simple as TEMP PROGRAM TO TEST SUBROUTINE LINKAGE. You may of course enter a version number or date, as well as a tape name or number. With the above label, this program would list just like this:

```
TEMP PROGRAM TO TEST SUBROUTINE LINKAGE
000     B     *R11
```

Here is a diagram for the Mini Memory Module. (It is described on the next page.)

DIAGRAM SHOWING LFAM TABLE, AND PROGRAM ENTRY POINT IN THE MINI MEMORY MODULE'S RAM



The First Program

As the arrows in the diagram on the previous page show, the computer locates your program by starting at LFAM (Last Free Address in the Module). From that point up to location 8000 is a list of one or more programs, each name having six characters, and with the entry point for each following the name in the next two bytes. You can store several programs in the module, calling each by name in whatever sequence you wish.

When the module is initialized with CALL INIT, LFAM is set to 8000. With one program, you must set LFAM to 7FF8. With two, set it to 7FF0. With three, set it to 7FE8. With four, 7FE0.

Looping

A strong point of computers is their speed. The use of loops makes it easy for the programmer to use this speed to do a task repetitively. In Basic, you should if at all possible use the FOR...NEXT construction for loops.

Below is a simple Basic program with a loop. If you enter the value of 1000 for N, the program should take about three seconds to complete. This means that each iteration of the loop requires .003 seconds (3 milliseconds).

```
100 INPUT N
110 FOR I=1 TO N
120 NEXT I
```

If this were to be translated directly into assembly language and told to loop 1000 times, it would be so fast you would not be able to detect any delay at all. Therefore, it is necessary to cause it to loop many more times. However, in assembly language one usually does not use the floating point numbers as Basic does. Instead, the computer's ability to handle 16 bit integer values is customarily used, but this means that a loop counter can only go as high as 32,767. In order to get a delay long enough to be measured, it is necessary to put a loop inside a loop, like this...

```
100 INPUT N
110 FOR I=1 TO N
120 FOR J=1 TO 1000
130 NEXT J
140 NEXT I
```

If you try this, give N the value 1. This will cause the inner loop to be executed 1000 times, so again it should take about three seconds. If N is 2, it will then take six seconds. This is the program that appears below in assembly language. It is called as a subroutine as shown in the previous chapter. This program is shown with the syntax for both the TI Editor/Assembler and the Dow Editor/Assembler.

There is a major difference between the Basic program above and the assembly language routine below. A language like Basic simplifies tasks such as entering data at the keyboard, so a convenient arrangement is to have Basic interact with the user by means of the keyboard and screen, and then have the assembly language program do the work. Therefore, the program gets N from a memory location where it was placed by the Basic program. This is easier than having the assembly language program prompt for the value N or having the Basic program pass N as a parameter. (The chapter on interrupts, screen, and keyboard describes in detail how to prompt for a value.)

Looping

To store a value into memory from Basic, you use the LOAD subroutine. Usually the value to be stored will be a word, which is two bytes. Because LOAD only stores bytes, it is necessary to break the word apart into two bytes. This is done with division, as follows:

```
BYTE1=INT(N/256)
BYTE2=N-256*BYTE1
```

For instance, if N were 15000, this is what would happen.

```
N/256                  58.59375
BYTE1=INT(58.59375)    58
256*BYTE1              14848
BYTE2=N-14848          152
```

Dividing N by 256 is the only method Basic can use to shift the left byte eight bits to the right. (Shifting is discussed in the chapter on shift instructions.) Because this results in a fraction, it is necessary to use INT. The right byte is obtained by shifting the left byte to the left again using multiplication and subtracting the result from the original value. Thus, the value 15000 consists of 58 in the first byte and 152 in the second. You can check this by 58 * 256 + 152 = 15000.

More difficult is determining the address to use. The first complication is the fact that when writing assembly language we use hexadecimal notation. However, Basic does not provide for hexadecimal constants. Therefore the argument for LOAD must be decimal.

As if the confusion between decimal and hexadecimal were not messy enough, it is necessary to treat hexadecimal addresses of 8000 and greater as if they were negative numbers. (If you have programmed other microcomputers in Basic, you are probably familiar with the use of PEEK or POKE with strange negative values; this is the same curse.)

Appendix C shows the listing of a program which converts between decimal and hexadecimal. Using this program is much easier than doing these conversions manually.

If you really want to know how to compute the appropriate decimal value manually, here is a brief explanation. You must first determine the two's complement of the hexadecimal value if it is at or above 8000. For example, if the address is A800, the complement is 5800. Convert the resulting value to decimal: the hex value 5800 = (5 * 16 + 8) * 256 = 22528. You then use the negative of that decimal value if the original hex value was at or above 8000. Thus, to load something into A800, use -22528 for the address. As another example, to load something into 7200, you would load into (7 * 16 + 2) * 256, which is 29184.

Looping

Here is the TI Editor/Assembler version of the assembly language looping program. Unlike the Dow Editor/Assembler version shown below, the TI version is loaded automatically, so it is not necessary to know exactly where it is located. The value of N can be passed to it through location A800, which is out in the middle of nowhere. (See Appendix A.) Remember that the ">" precedes hexadecimal values.

```
        DEF  TEMP
TEMP    MOV  @>A800,R0     LOAD R0 FROM LOCATION A800
LP1     LI   R1,1000       SET R1=1000
LP2     DEC  R1            R1=R1-1
        JGT  LP2           LOOP IF R1>0
        DEC  R0            R0=R0-1
        JGT  LP1           LOOP IF R0>0
        RT                 RETURN
        END
```

This Basic program will load and run the TI version. It is written to pass any value of N to the subroutine. However, it does not check the value, so make sure you only enter a positive integer value between 1 and 32,767.

```
100 CALL INIT
110 CALL LOAD("DSK1.TEMPOBJ")
120 INPUT N
130 BYTE1=INT(N/256)
140 BYTE2=N-BYTE1 * 256
150 CALL LOAD(-22528,BYTE1,BYTE2)
160 CALL LINK("TEMP")
```

Note that LOAD in statement 110 loads the object program from disk into memory, while the LOAD in statement 150 stores specific values (BYTE1 and BYTE2) into memory at a specified location.

Start timing after entering the value of N. Statements 130 and 140 break the value apart into two bytes, and statement 150 stores the two bytes into locations A800 and A801.

Here is the Dow Editor/Assembler version of the program. It is loaded at 7118, the first free location in the module, and it gets the value of N from location 7200.

```
     MOV   @>7200;R0     LOAD R0 FROM LOCATION A800
LP1:LI     R1;1000       SET R1=1000
LP2:DEC    R1            R1=R1-1
     JGT   LP2           LOOP IF R1>0
     DEC   R0            R0=R0-1
     JGT   LP1           LOOP IF R0>0
     B     *R11          RETURN
```

To pass the value 1000 to "N" to the program, you could use a Basic program to prompt for N and store it into memory, very similar to the Basic program shown above for the TI version. However, it is easier to follow the steps below. The advantage is that you can leave the Dow Editor/Assembler running while you test your assembly language program. Start the timing after entering the LINK command. (The "hhhh" below merely represents the value in location 7200, whatever it may be.)

```
---->MINI 7200
>7200 >hhhh ?1000
>7200 >03E8 ?.
---->LINK TEMP
```

At this point, you should have been able to enter, assemble, and run this program using one of the editor/assemblers. Let us now examine the program one statement at a time.

The purpose of explaining these statements here is only to give you a better feel for assembly language. (Each statement will be described again later in the book in the chapter which discusses the appropriate instruction types.) Do not be discouraged if you do not understand everything at this time.

DEF - This appears only in the TI Editor/Assembler version. Its purpose is to define an entry in the REF/DEF table so the CALL LINK in the Basic program will be able to transfer con- trol to it. This statement was described earlier in this book and will be discussed again in the chapter on directives.

MOV - The value of N was stored in memory location A800 or 7200, depending on which editor/assembler you used. This instruction moves it from that location into register 0. Thus, if you entered the value 500 for N with the MINI command or with the Basic program, it would now be placed into register 0.

LI - This is a special form of move. Load immediate moves a value from right there in the program (actually a part of the instruction itself) into a register. Whereas MOV is roughly equivalent to the Basic statement LET X=Y, LI is similar to LET

X=9. In this program, the LI statement sets register 1 equal to 1000. Register 1 is the counter for the inner loop.

DEC - Now begins the inner loop. This instruction means to decrement register 1. Since it was just set to 1000, it is now 999. Each time through the loop, register 1 will be decremented by 1. It thus fulfills its role as the loop counter. Unlike the typical loop counter in Basic, it counts down toward 0, not up from 1.

JGT - This statement completes the loop. It means to jump if greater than to the statement with the label LP2.

At this point, a review of the parts of a loop in Basic is needed for comparison to the assembly language version. A loop has a variable which is the counter. That variable is: 1) given an initial value; 2) incremented (or decremented, as with STEP -1); 3) possibly used within the loop for other purposes (such as an index into an array); and 4) tested to determine when to leave the loop.

In the assembly language loop, these same parts exist. However, you have to explicitly put them there. Usually, a register serves as the loop counter instead of a memory location (the counter part to a variable in Basic). In the inner loop, the counter is register 1. It is: 1) initialized with LI to be 1000; 2) decremented by DEC; 3) not used within the loop, although it could be; 4) tested by the JGT to determine when to leave the loop.

To complete the explanation, it is important to cover how JGT can tell when to stop looping. The answer is that on the TI, as well as on many computers, it is standard to provide automatic or simple tests against the value 0. Such tests are possible for registers, memory locations, and the result of various operations (such as adding or subtracting, incrementing or decrementing, or even just moving). For example, the DEC instruction on the TI not only decrements the register, it also tests the new value against 0. The result of the comparison is left in certain bits in the status register (one of the TI's hardware registers described earlier). These bits are tested by instructions such as JGT. Thus, the JGT really means "if the most recently performed operation which did a comparison resulted in a value greater than 0, then jump".

This automatic comparison to 0 is why loops in assembly language count down to 0. For example, instead of counting down by 1's with DEC, it would certainly be possible to count up by 1's with INC (which is the increment-by-1 instruction). However, in that case you would have to insert an extra statement in the pro-

gram to compare the count to the final value. This comparison instruction would be placed before the jump, at the end of the loop. Including this instruction would make the program a little larger and a little slower, so the natural tendency of assembly language programmers is to take advantage of the automatic capabilities of the computer and decrement to 0 when possible.

Because the decrement comes before the test, the value in register 1 during the last iteration through the loop will be 1. After leaving the loop, the register will contain 0. If you want the last value to be used in the loop to be 0, use the JOC instruction instead of the JGT instruction. (Both JGT and JOC are described again in the chapter on jumps.)

Arithmetic Operations

Computers, as the English language name implies, can compute. For many applications, the computations are actually quite trivial mathematically. For instance, computers frequently count things, or look in lists, or find the middle or average. These operations only require simple arithmetic operations, frequently with integer values.

Programs written in Basic on the TI are capable of handling large numbers with great precision. The language also provides advanced arithmetic functions, such as the trig function SIN. However, most programs don't use any of these features. Instead, the integer arithmetic capabilities of the computer itself, the 9900 processor, are very adequate. These operations include addition, subtraction, multiplication, and division.

All of these operations are "binary", as opposed to "unary". That means that they operate on two values. On the TI, the operands (the values operated on) can be registers or memory locations. Here are examples of the binary (that is, two operand) arithmetic operations, **addition** and **subtraction.** Examples of unary operations will come later in this chapter.

```
A    R1,R12      add the contents of register 1 to
                 the contents of register 12
A    @COUNT,R5   add COUNT to register 5
S    R3,@ABC     subtract register 3 from ABC
```

For all these intructions, the first operand is called the **source** and the second is the **destination.**

In performing an addition, the sum of the two values replaces the second value. The destination becomes the sum of the source and destination. Thus, read "A @X,@Y" as "add X to Y"; this is similar to "Y=Y+X" in Basic.

For subtraction, the difference between the two values replaces the second value. The destination becomes the difference between the destination and source. Read "S @X,@Y" as "subtract X from Y"; in Basic, "Y=Y-X".

Here are examples of **multiplication** and **division.**

```
MPY R5,R0        multiply register 5 times register 0
                 (product in registers 0 and 1)
MPY @SCALE,R9    multiply SCALE times register 9
                 (product in registers 9 and 10)
DIV R1,R2        divide register 1 into registers 2 and 3
                 (quotient in register 2, remainder in 3)
```

Arithmetic Operations

Since multiplication and division are somewhat different in Basic and assembly language, actual Basic statements cannot be used to show how they work. Therefore, in the discussion below you will see statements that look something like Basic but are not. However, the differences are simple and do not have to be explained formally. The examples are provided merely as something roughly familiar in order to facilitate understanding.

The format of the multiplication instruction differs from the addition and subtraction instructions in that the destination of a multiplication must be a register, whereas for the other two that is just one of the options. The operation itself differs in that the result of a multiplication may actually be a two-word value. That is, the result occupies two registers, not just one.

This size doubling during multiplication is true of multiplication in any number system. For instance, multiply two 3-digit numbers and the product may have as many as six digits. As an example, 999 times 999 is 998,001.

The instruction "MPY @X,R1" multiplies X times register 1, putting the result into registers 1 and 2 combined. The two 16-bit values result in a 32-bit value. Read it as "multiply X times register 1, putting the result into registers 1 and 2." In pretend Basic, this might be "R1c2=X*R1". (I am using "R1c2" to mean registers 1 and 2 combined.)

If the multiplier and multiplicand will always be small enough, you will know that the result of a multiplication operation cannot in fact be a two word value. In those cases, the first register will always be 0 and you can write the program to just use the result left in the second register.

Here are the listings for two programs which will let you experiment with the multiplication instruction. This should help you understand how it works. The first program is in Basic and the second is an assembly language subroutine. The Basic program prompts for two values, then calls the assembly language routine to multiply them together. Store the Basic program in the file "DSK1.TESTB" and the assembly language program in the file "DSK1.TESTA". The object program should be stored in "DSK1.TESTAOBJ". (Instructions will follow for using the the Dow Editor/Assembler.)

Here is the Basic program.

```
100 REM TEST ARITHMETIC INSTRUCTIONS (DSK1.TESTB)
101 CALL INIT
102 CALL LOAD("DSK1.BSCSUP","DSK1.TESTAOBJ")
110 LOC=-22528
120 FOR I=0 TO 2 STEP 2
```

```
130 INPUT N
140 IF N>-1 THEN 160
150 N=N + 65536
160 BYTE1=INT(N/256)
170 BYTE2=N-256 * BYTE1
180 CALL LOAD(LOC+I,BYTE1,BYTE2)
190 NEXT I
200 CALL LINK("A")
210 FOR I=0 TO 6 STEP 2
220 CALL PEEK(LOC+I,BYTE1,BYTE2)
230 N=BYTE1 * 256 + BYTE2
240 IF BYTE1<128 THEN 260
250 N=N-65536
260 PRINT "(";
270 B=BYTE1
280 GOSUB 360
290 B=BYTE2
300 GOSUB 360
310 PRINT ") ";N
320 NEXT I
330 CALL PEEK(LOC+4,BYTE1,BYTE2,BYTE3,BYTE4)
340 PRINT ((BYTE1 * 256 + BYTE2) * 256 + BYTE3) * 256
+ BYTE4
350 GOTO 120
360 REM BINARY TO HEX CONVERSION
370 HD1=INT(B/16)
380 HD2=B-16 * HD1
390 S$="0123456789ABCDEF"
400 PRINT SEG$(S$,HD1 + 1,1);SEG$(S$,HD2 + 1,1);
410 RETURN
```

This is the assembly language routine.

```
      TITL 'TEST MULTIPLY INSTRUCTION' (DSK1.TESTA)
      DEF  A
A     MOV  @>A800,R0   LOAD R0 FROM LOCATION A800
      MPY  @>A802,R0   MULTIPLY VALUE IN A802 TIMES R0
      MOV  R0,@>A804   PUT LEFT HALF OF PRODUCT IN A804
      MOV  R1,@>A806   PUT RIGHT HALF IN A806
      B    *R11
      END
```

If you try these programs and enter 3 and then 7 in response to the prompts, you will see the following output.

```
(0003)  3
(0007)  7
(0000)  0
(0015) 21
 21
```

The Basic program stores your two values in the words starting at

locations >A800 and >A802. The assembly language program multiplies those two values and puts the result into the two words starting at locations >A804 and >A806. These four words are displayed by the Basic program as 3, 7, 0, and 21. (The values in parentheses are the same values in hexadecimal notation.) The value 21, shown on the last line, is the result of accumulating all four bytes of the product into one value by successively multiplying by 256 and adding.

As discussed above, the product occupies two words. For a small positive product such as 21, the first word is 0. Now try a negative value to see what happens.

```
(0003)  3
(FFF9) -7
(0002)  2
(FFEB) -21
 196587
```

As a result of taking the product of 3 and -7, the first word of the product is always 1 less than the positive value, or 3 - 1 = 2 in this case. The second word contains the expected -21. The two words together yield 196,587, which is useless. Therefore, just as you can multiply small positive values and ignore the first word of the product, so also you can multiply small negative values and ignore the first word of the product. However, if large values must be multiplied, be sure both are positive before doing so.

Try one more example. This is what happens when the two values to be multiplied are large enough to create an actual two-word product. The two values are 10,000 and 10,000 and the product is 100,000,000. In this case, neither the first word of the product nor the second is meaningful by itself.

```
(2710)  10000
(2710)  10000
(05F5)  1525
(E100) -7936
 100000000
```

Division also requires the second operation to be a register. Just as with multiplication, it is actually two registers combined. A two word value is divided by a single word value to get a single word result. That is, a 32-bit value is divided by a 16-bit value to get a 16-bit result. There is also a remainder, which is also a single register (16-bits).

Although multiplying two 3-digit numbers cannot yield a product with more than six digits, dividing a 6-digit dividend by a 3-digit divisor may yield a result of more than three digits. This is an **overflow** condition that is detected by the computer.

For example, divide 300 into 354,137 and the result is more than three digits. In many cases, this poses no problem for your programming because you should know in advance if such a situation will arise with your data. For instance, if you divide 537 into 354,137 you get 659 with remainder 254.

Division in Basic may result in a number with a fraction. For instance, in Basic, 354,137 divided by 537 is 659.47299. The division instruction in assembly language cannot yield a fractional number because all numbers are integers. Therefore, if there is anything left over it must be expressed as a remainder instead of a fraction. This is how you can compute a remainder manually.

```
354,137 divided by 537      659.47299
take the integer result     659
multiply 537 * 659          353,883
subtract 354,137 - 353,883  254
```

Even though you want to divide a single word value such as 10,000 by another single word (but smaller value), you still need a 2-register dividend. To do this, you must put the dividend into the second of two registers and make sure the first is 0. You can clear a register with the CLR instruction; thus, to zero register 1, use "CLR R1".

Read "DIV @X,R9" as "divide X into registers 9 and 10, with the result in register 9 and the remainder in register 10." In pretend Basic, you might write "R9=INT(R9c10/X) and R10=remainder(R9c10/X)".

Try this assembly language program to become familiar with the division instruction.

```
          TITL 'TEST DIVIDE INSTRUCTION' (DSK1.TESTA)
          DEF  A
A         MOV  @>A802,R1   LOAD R1 FROM LOCATION A802
          CLR  R0          SET R0 TO 0
          DIV  @>A800,R0   DIVIDE CONTENTS OF A800 INTO R0,R1
          MOV  R0,@>A804   STORE QUOTIENT IN A804
          MOV  R1,@>A806   STORE REMAINDER IN A806
          B    *R11
          END
```

If you divide 5 into 100, this will be displayed.

```
(0005)  5
(0064)  100
(0014)  20
(0000)  0
 1310720
```

This shows that 5 into 100 gives 20, remainder 0. (The value on the last line is a meaningless combination of the quotient and remainder and should be ignored.)

Now divide 5 into 103 to see what happens when there is a remainder.

```
(0005)  5
(0067)  103
(0014)  20
(0003)  3
 1310723
```

Try some negative values with the DIV instruction. You will undoubtedly conclude that it is better to try to stay with positive values. This should not be a problem too often. If it is, you will have to test for negative values, then use ABS to make it positive, and use NEG to make it negative later.

Remember that both the multiply and divide instructions operate on **unsigned** numbers. These are numbers that are treated as positive values, even though the left-most bit, the sign bit, may be a 1. Try the test programs above to understand what happens when negative values are used.

Let us summarize the binary operations. It should be easy to remember both the addition and subtraction instructions. Multiplication is similar to addition and subtraction, except that the second operand must be a register and the result is two words. Division is the most complicated. Like multiplication, the second operand is a register combination, but it starts out as two registers together and ends up as two separate registers. The most difficult thing to remember is which is which; perhaps it will help to remember that they are in alphabetical order - the first is the Quotient, and the second is the Remainder. Another scheme for remembering is to think that the quotient is used more often than the remainder, so it is put into the first register.

There are several other arithmetic instructions. In addition to addition and subtraction on words, there are also **byte addition** and **byte subtraction** instructions, AB and SB respectively. These are exactly the same as A and S (for words), except that they add bytes. (Remember that if a register is an operand, only the left half is used.)

Arithmetic Operations

There is a special addition instruction, called **add immediate**. The term "immediate" means that the value to be added is right there in the instruction. Unfortunately, the order of the operands is reversed from the normal addition instruction so that the sum replaces the first operand, not the second. The instruction looks like this.

```
AI  R5,980      add 980 to register 5
AI  R0,-100     add -100 to register 0
```

There is no "subtract immediate" instruction. However, you can add the complement of the value you want to subtract. For example, to subtract 100 you should add -100. See the example above.

There is no immediate addition or subtraction for bytes, and no immediate multiply or divide.

There are four instructions which do a special form of arithmetic. (All are instances of **unary** operations, while all operations discussed so far were binary.) Two of these instructions add or subtract the value "1", and two add or subtract the value "2". These are called **increment** and **decrement**, and are very useful when writing loops, as discussed in the last chapter. Here are examples of these special addition and subtraction instructions.

```
DEC  R3     Subtract 1 from register 3.
DECT R9     Subtract 2 from register 9.
INC  @SW    Add 1 to SW.
INCT R0     Add 2 to register 0.
```

The reason there are instructions which add or subtract two is that frequently a program operates on a list of word values rather than on byte values, and since the memory is addressed by bytes, each word value is two bytes from the last.

Although these instructions are often used to increment or decrement loop counters, they can be used whenever you simply want to add or subtract one or two. Another use can be for setting register or memory locations which you are using as switches. For instance, you can use CLR R5 to set register 5 to 0, and then you can use INC R5 to set it to 1.

Finally, there are two remaining arithmetic instructions, **absolute value** and **negation**, both of which only can be used on words (not bytes).

The first, **ABS**, sets the register or location to a positive value but leaves it alone if already positive. For instance, if location DIFF contained the value -193, then ABS @DIFF would set the value to +193.

The second, **NEG**, sets the value in a location or register to its complement. Thus, if the value is negative, it is set positive, and if positive, it is set negative. For example, if register 3 contained +5, NEG R3 would set it to -5, whereas if it contained -5 to begin with, NEG R3 would set it to +5.

The test programs shown above are for the TI Editor/Assembler. The value -22528 given to LOC in the Basic program statement 110 is equal to hex A800. Below is what the assembly language program looks like when listed from the Dow Editor/Assembler and the Mini Memory Module. Notice that >7200 is used instead of >A800, that >7202 is used instead of >A802, and so forth. Load the program at >7118. It is also necessary to put the label A and the address >7118 into the REF/DEF table. (Go back to the chapter "The First Program" to see how to do this.) In the Basic program, set LOC in statement 110 to 29184 (which is equal to >7200) and delete statements 101 and 102 from the Basic program.

```
TEST MULTIPLY INSTRUCTION
000     MOV  @>7202;R0
004     MPY  @>7200;R0
008     MOV  R0;@>7204
00C     MOV  R1;@>7206
010     B    *R11
```

You may wish to modify these programs to try other instructions, such as addition and subtraction on words or bytes. This is a useful exercise to get a good feel for 2's complement notation.

Addressing Modes

Nearly every instruction refers to some data in a register or in memory. The act of referring to the data is called **addressing.** The way in which it does it is called the **addressing mode.** A diagram at the end of this chapter shows most of the modes in simple schematic form.

So far in this manual, you have seen data accessed both in registers and in memory locations. These are just two of a number of ways of addressing data. Although I have described instructions as only allowing reference to registers or memory, there are several other modes also available for most of those instructions. The examples below will show how this is done.

The first addressing mode to be discussed in this chapter is **register addressing,** which means that the value to be used is in the specified register. A register is simply a word in memory that can be addressed easily by a number in the range 0 to 15. (Remember that TI calls these **software registers.**) On some computers, registers are not memory locations; instead, they are **hardware registers.** Because they are implemented in hardware rather than in memory, they can be much faster than memory. Of course memory could be made as fast as a hardware register, but this would make the computer very expensive.

However, even though a register on the TI is not as fast as on machines with hardware registers, the use of registers can represent a speed gain over the use of memory addressing (to be discussed below). This is because the register is specified by 4 bits within the instruction itself, whereas a memory reference requires that the instruction include an additional word to contain the 16 bit address, which in turn points to the value in memory. Because the processor has to load both the instruction and the address word from memory, the instruction will take longer to execute. The extra word also makes the program larger.

Note that the 256 bytes of RAM within the console is faster than the memory in the Mini Memory Module or the 32K memory expansion. Typically the registers are located in this 256 byte memory whereas variables you define in your programs are much more apt to be in the slower memory. This means that even the software registers on the TI are faster than the usual memory addressing.

Here are some examples of register addressing.

```
ABS   R0      Take absolute value of register 0.
A     R1,R9   Add register 1 to register 9.
MOV   R9,R3   Move register 9 to register 3.
AB    R6,R1   Add the left byte of register 6 to the
              left byte of register 1.
```

Direct addressing, or **symbolic memory addressing**, or just **memory addressing**, means that the address is part of the instruction (as an additional word). The address points to the memory location whose contents are to be used by the instruction. This address is an **absolute address**. Here are two examples.

```
ABS   @>7200  Take the absolute value of locations 7200
              and 7201 (hexadecimal).
ABS   @XYZ    Take the absolute value of locations XYZ
              and XYZ+1.
```

In the first example, the contents of the word at locations >7200 and >7201 is used. The address could be specified in hexadecimal or decimal, although hexadecimal is more usual.

In the second example, locations XYZ and XYZ+1 are used. Since "XYZ" is a symbol, by the definition of a symbol it represents something other than itself. You cannot tell by looking at the instruction just what memory location is to be used. There must be some other place in the program where XYZ is defined.

One way to define a symbol such as XYZ is to use it as a **label** for some data. Example:

```
XYZ    DATA  906
```

(The DATA directive will be discussed in a the chapter on directives.)

Another way to define a symbol is with the **EQU** (equate or equivalence) directive, like this:

```
XYZ    EQU   >A000
```

This would mean that wherever the symbol XYZ has been used in the program, you could just as well have used >A000.

Addressing Modes

With the Dow Editor/Assembler, the symbol must be EQU'd to a decimal or hexadecimal constant. Remember that it is best to put the EQU's at the end of the program.

However, with the TI Editor/Assembler, the symbol can be EQU'd to other symbols or even to the result of certain computations. Here is a more complicated example:

```
XYZ    EQU   BUFSTR+100
```

This sets XYZ to 100 (decimal) greater than the value of BUFSTR. This type of computation is useful when there must be a specific relationship between two symbols in memory.

A symbol can also be given a value with the **REF** directive, but this only works with the TI Editor/Assembler loader and not with the Dow Editor/Assembler or with Extended Basic. Here is an example of defining the symbol VMBW for use with the three techniques of running an assembly language program. (VMBW is the name of a utility routine that is used to display on the screen.)

```
1)         REF  VMBW    TI Editor/Assembler loader.

2) VMBW    EQU  >2024   Extended Basic loader

3) MBW:    EQU  >6028   Dow Editor/Assembler
```

In case 1, the program is loaded with the TI Editor/Assembler module in place. The programmer does not need to know the exact location of the utility routine. In the second case, the program is assembled using the TI Editor/Assembler, but it is loaded with the Extended Basic module in place, so the programmer must supply the correct address. These addresses are listed in section 24.4.8 of the TI Editor/Assembler manual. The third case is essentially the same as the second, although the module in place is the Mini Memory Module and the location is different. These locations are listed in the Mini Memory manual, starting on page 35.

Another form of memory addressing uses both memory and register. **Indexed memory** mode uses a memory address, as described above, but also includes a register. The processor adds the memory address specified in the instruction to the contents of the register specified in the instruction; the resulting value, called an **effective address**, points to the data in memory. Here is an example.

```
A      @LIST(R1),R5
```

This means to use register 1 as an index into a list named LIST, adding the value of the appropriate item to the value in register 5. This is very similar to the Basic statement "R5=R5+

LIST(R1)". That is, indexing in assembly language is like subscripting in Basic (or mathematics).

To help clarify what indexing is, assume that the symbol LIST is equivalent to >A000 and that register 1 contains 20 decimal, or >14 hexadecimal. The instruction above would refer to the word value at locations >A014 and >A015. (Remember that word values must start at even locations.) If register 1 were to be incremented by 2, the instruction would refer to the word value at locations >A016 and >A017. Thus, if there is a list of word values pointed to by LIST, this single statement can add them all into register 5.

Indexing is used frequently in loops. If you are refering to word values, the loop counter should be incremented or decremented by 2's, not by 1's. That is, use INCT and DECT, not INC or DEC. Also, if the index value is derived somehow - perhaps the value of the key pressed by a user is to be used to look up a value in a list - the value must be doubled before being used as an index. (You can easily double a value by adding it to itself, as "A R1,R1".)

A small note of caution regarding indexing is necessary. You cannot use register 0 as an index register. The reason for this is very simple: the direct memory address mode and the indexed memory mode are stored in the same fashion internally, with the direct mode appearing to index by register 0. The computer acts as though register 0 always contains the value 0.

Another mode of addressing is **indirect addressing.** There are two forms. Both use registers and are indicated by an asterisk before the register. Here is an example of the first form.

```
A      *R4,R5     Add indirect register 4 to register 5.
```

Register 4 does not contain the value to be added, but contains the address of the value in memory which is to be added. Register 4 is in effect a **pointer.**

The closest Basic comes to a pointer is the PEEK subroutine. If you use the statement "CALL PEEK(LOC,A)" the contents of the location pointed to by LOC will be moved into the variable A. Pretend that Basic has a function WORDPEEK(LOC) which returns the value of the word pointed to by LOC. Then "R5=R5+WORDPEEK(R4)" would be the same as the assembly language instruction above.

The second form of indirect addressing, **register indirect auto-increment**, is very similar. This is what it looks like.

```
A      *R4+,R5    Add indirect register 4 (incremented) to
                  register 5.
```

The use of the "+" means that the contents of the register, the pointer value, is to be incremented after the instruction is executed. Whether it is incremented by 1 or by 2 depends on whether the instruction refers to a byte value or a word value. (Byte instructions are identifiable by their names: eg, AB is add byte.)

The auto-increment feature is very useful in loops, similar to the way that indexing is. There are differences, however. For instance, you may not know when you write the program where the list will be located in memory so you cannot have a label for it such as LIST in the example above. In that case, you can make a register point to the beginning of the list, and the computer will automatically move the pointer along the list as you go through the loop. Since it will increment by 2's if you are processing word values, you can use a loop index which increments only by 1's without having to double it to use it as an index value. An additional benefit of this mode is the same advantage that register addressing has over memory addressing - namely that it is not necessary to include the address in the instruction, which makes the program a little smaller and faster.

Program counter relative addressing on the TI is only used for jump instructions. Here is a typical jump instruction.

```
JMP  TOP    Jump to statement with label TOP.
```

On some other computers, this mode can be used for instructions such as add or move. It is a special form of memory addressing. The nice thing about it is that the address is contained within the instruction itself (unlike a normal memory address but like a register address). The bad thing is that it can only refer to memory locations within the vicinity of the instruction itself. The way it works is somewhat similar to indexing, in which a register value is added to a memory address. With PC relative addressing, the memory address that is used is the address of the instruction itself, and instead of the value of a register, the value of the **displacement** (stored within the instruction) is added. In effect, a jump says "jump 20 locations down in the program," or "jump 180 locations back up in the program." There is nothing in the instruction which specifies an absolute address, which is why it is called "relative" addressing.

The reason this type of addressing is nice is that an instruction can transfer control to another location without having to use a second word to point to the new location. The displacement value is the byte in the right half of the instruction itself. This means that it has 8 bits, so it can have 256 possible values. These are treated as -128 through +127. Because instructions are all an even number of bytes in length, the value stored in the instruction is doubled before being used. Also, the value is added to the PC after the instruction has been

executed, so if you wanted to jump to the next instruction you would end up with a displacement of 0 because the PC has already been incremented by the processor to point to the next instruction.

Although the instruction contains a numeric displacement value, when you look at an assembly language program you see a label instead. In the example above, the statement with the label TOP could be either before or after the jump, provided it falls within the -128 and +127 word limit. The assembler computes the difference between the instruction location and the labeled statement and puts the correct value into the instruction.

With the TI Editor/Assembler, you have another option with PC relative addressing which should in fact be avoided. If used at all, the jump should be to a location not very far away because, as you will see, you have to count how far to jump and it is easy to count incorrectly and thereby cause an error. Furthermore, if you change your program so that an instruction is added or deleted, the count will be wrong and the jump will go to the wrong location. This type of bug can be very difficult to locate.

Now that you have been cautioned not to use this form, this is how you do it. You explicitly refer to the relative nature of the addressing. The symbol $ is used to mean the current location, so "JMP $+10" would mean to jump down five words, or 10 locations. The value you specify should be an even value, since the assembler will divide it by two because when stored in the instruction, it refers to a word count, not a byte count.

If you want to, with the TI Editor/Assembler you can specify an absolute location with a jump instruction. In that case, the assembler has to convert it into a relative address anyway. The same restrictions about how far it is possible to jump apply in this case.

Immediate addressing is a mode which was described in the context of the **add immediate** instruction in the chapter on arithmetic. In this mode, the value to be used is present as part of the instruction. All instructions with immediate addressing on the TI indicate that fact by their name, and all are two word instructions. The immediate instructions are: AI, CI, LI, and LIMI.

Addressing Modes

Remember that for AI, CI, and LI, the first operand must be a register and the second an immediate value. Look at these four examples. The error messages from the TI Editor/Assembler and the Dow Editor/Assembler are shown.

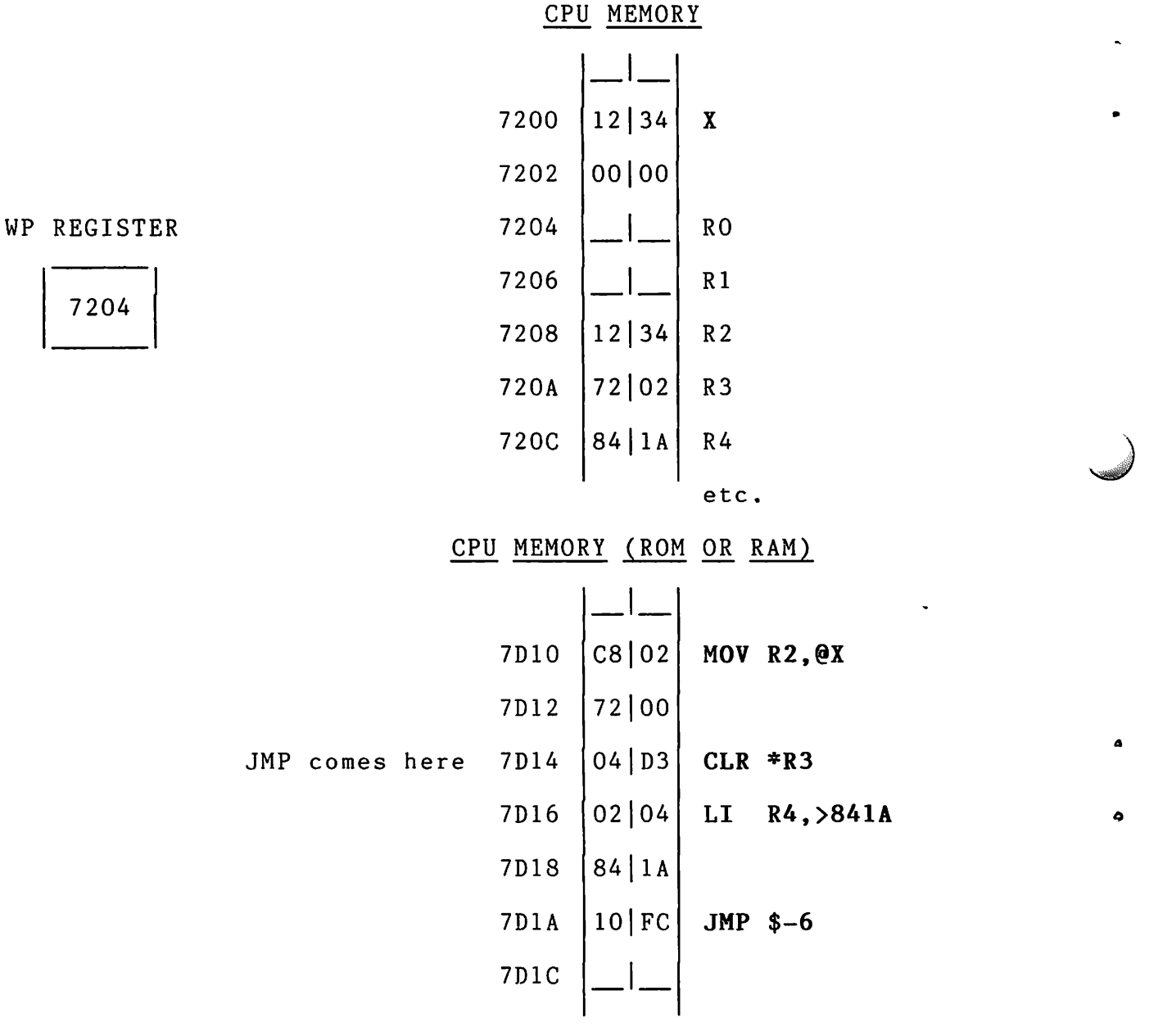
	INSTRUCTION		TI E/A MESSAGE	DOW E/A MESSAGE
1)	CI	@A,29	SYNTAX ERROR	ERROR @ ^
2)	CI	A,29	INVALID REGISTER	ERROR A ^
3)	CI	R2,4	no error	no error
4)	CI	R2,R6	no error	ERR: LBL R6 AT loc (shown during LOAD)

Notice that in one case the TI Editor/Assembler will not detect an error that the Dow Editor/Assembler will detect. This happens because the Dow Editor/Assembler knows that the symbols R0 through R15 are always reserved for registers. On the other hand, the TI Editor/Assembler simply maps these symbols into the values 0 through 15; therefore, if you intend such a symbol to mean a register even though register usage is inappropriate in the particular case, it will be assembled with no error message if the corresponding value is acceptable.

Finally, the TI communicates with the outside word (such as printers, phone modems, and disk drives) through the **CRU** (Communications Register Unit). Instructions which refer to the CRU include values which are CRU bit addresses. These will not be discussed further here.

Here is a simple diagram which shows most of the addressing modes.

DIAGRAM OF ADDRESSING MODES



The MOV instruction shows the register mode with "R2" and the memory mode with "@X". The contents of register 2 are moved to location X. Register 2 is at location 7208 because the WP register contains 7204. X is defined to be location 7200 (perhaps with "X EQU >7200"). Notice that 7200 is stored as the second word of the instruction. The instruction moves the value from locations 7208 and 7209 to locations 7200 and 7201. (In the

example, the value is "1234".)

The CLR instruction shows indirect register addressing. Register 3 is at location 720A, where the value 7202 has already been stored. Therefore, the instruction clears location 7202 and 7203. (The diagram shows the 0's resulting from the CLR instruction.)

The LI instruction shows the immediate mode. The value to be moved into register 4 (at location 720C) is stored as the second word of the instruction. Compare the second word here to the second word of the MOV - with memory addressing, a word is appended to the instruction to hold an address, while with immediate addressing, a word is appended to hold a value.

The JMP shows PC relative addressing. The amount to jump is stored within the right half of the instruction. To compute the value, first subtract 2 from the amount specified in the assembly language statement. Thus, subtract 2 from -6 to get -8. (This is done because the PC register will have been incremented by two bytes before the arithmetic on its value is performed by the CPU.) Then divide the result by 2; -8 / 2 = -4. If negative, convert this to two's complement; -4 = FFFC. Finally, take the right half of the result, FC.

Remember that an instruction such as JMP $-6 can be poor programming, since it is often better to use labels. The Dow Editor/Assembler does not even support this form. With either assembler, if a JMP with a label is used, it generates PC relative addressing anyway.

Jump and Branch Instructions

These instructions are all used to alter the flow of a program. There are several classes of these instructions: **unconditional, conditional, subroutine calls, execute a remote instruction,** and **extended operation.**

A major difference between Basic and assembly language is that the statement numbers in a Basic program are used by GOTO statements, but statement numbers in assembly language programs are only to number the lines for reference purposes. Jumps and branches in assembly language typically use statement labels. (There are other methods, but labels are preferable.) Look back in the chapter on looping to see examples of statement labels: LP1 and LP2.

Unconditional jumps or branches in assembly language perform the same function as GOTO statements in Basic. There are two assembly language instructions: **JMP** and **B.** The difference between the two is that the B (for branch) can go to a location anyplace in memory, but the JMP (for jump) only uses the **PC relative mode** of addressing, meaning that it can only jump a relatively short distance to a statement with a label. (See the chapter on addressing modes - there are alternative forms, but they should be avoided.) Here is an example.

```
JMP  TOP
```

The JMP instruction is appropriate for most of the times you want to transfer control unconditionally, even though a branch instruction can go longer distances. The reason to prefer the jump is that a well written program consists of modules or segments within which you have gathered logic pertaining to a particular process. As discussed below, control is passed to such modules with either of the BL or BLWP instructions, so the JMP is only needed within modules.

The **B** instruction is far more flexible than the JMP but in turn is overkill if you only want to move forward or backward in the program a small amount. If you were to use B to go to a label, it would look like this.

```
B    @TOP
```

This would do exactly the same as the jump shown above, but it requires two words of memory instead of just one.

The real usefulness of B over JMP is in the flexibility of addressing that is possible with B. For instance, you can branch to a location which is held in a register. This is how to return from a subroutine called with BL. You branch indirect of regis-

ter 11, in which the return address was automatically stored by the BL.

```
        B    *R11
```

However, B can also be used in a way similar to the ON GOTO of Basic. Suppose you want to branch to one of three labels in your program, depending on an integer value. You would have to double the value for use as a subscript into a list of label addresses

The following example assumes that the value to be used for the branch is passed from the program DSK1.TESTB into location >A800. The program returns one of the values >0000, >1111, or >2222 in location >A802 to prove that it branched correctly. Notice that it is necessary to move the address from the vector into a register in order to branch indirect, since there is no way to branch indirect of a memory location.

```
        TITL 'IMPLEMENT ON GOTO' (DSK1.TESTA)
        DEF  A
A       MOV  @>A800,R1      Move value into register 1.
        A    R1,R1          Double it for word addressing.
        MOV  @VEC(R1),R1    Move branch address into register 1.
        B    *R1            Branch indirect of register 1.
VEC     DATA L1             List of branch addresses.
        DATA L2
        DATA L3
L1      LI   R1,>0000       Value was 0. Return 0000.
        JMP  END
L2      LI   R1,>1111       Value was 1. Return 1111.
        JMP  END
L3      LI   R1,>2222       Value was 2. Return 2222.
END     MOV  R1,@>A802
        B    *R11
        END
```

The **conditional** jumps are used for the equivalent of the IF in Basic. Unfortunately you cannot simply say as you can in Extended Basic "IF X>Y THEN Z=3". You cannot even say the simpler "IF X>Y THEN 300". Instead, in assembly language it is necessary to separate the operation which performs the comparison of X and Y from the operation which alters the flow in the program based on this comparison. The comparison itself can be done either with a special compare instruction, or it can depend on numerous other instructions which automatically compare a value against 0. In either case, the result of the comparison sets some bits in the **Status Register.** There are then a number of jumps which you can use to complete the "IF" construct because they jump conditionally. For instance, if you want to know if "X>Y", you would compare X and Y, then JGT, like on the next page.

```
        C     @X,@Y     Compare X and Y.
        JGT   S1        Go if X>Y.
```

In this example, S1 is the label to which control will transfer if X is greater than Y.

In addition to JGT (jump greater than), there is also JLT (jump less than), JEQ (jump equal), and JNE (jump not equal). With combinations of these you can perform all the arithmetic comparisons you need.

Let us examine again the example above: IF X>Y THEN Z=3. It is actually inappropriate to jump if X is greater than Y, since if that condition is true we want to do something, but if it is false we don't want to do it. So really, the jump should take place if X is less than or equal to Y. Unfortunately, there is no such jump instruction. However, it can be done with two jumps, as follows.

```
        C     @X,@Y     Compare X and Y.
        JLT   S1        Go if X<Y.
        JEQ   S1        Go if X=Y.
        MOV   @V3,@Z    Set Z=3.
S1      (program continues here)
        ...
        (The following statement must also
        appear in the program.)
V3      DATA 3          The value 3.
```

This last example shows one of the problems you will encounter when programming in assembly language. Often you will have to use a jump which is the opposite of what you want to do, and the opposite of what you would say in Extended Basic. Actually, the problem is very similar in Console Basic, in which the same example would have to be programmed something like this:

```
100 IF X<=Y THEN 120
110 Z=3
120 (program continues here)
```

There is another set of jumps which are not based on **arithmetic comparisons** but instead are based on **logical comparisons.** These are JH (jump high), JL (jump low), JHE (jump high or equal), and JLE (jump low or equal).

The jumps JEQ and JNE may be considered as either arithmetic or logical. There is no arithmetic equivalent of the logical JHE and JLE.

In general, the arithmetic jumps first discussed would be used when doing calculations and testing numeric ranges, while

the logical jumps would be used when comparing character codes for sorting.

Here is an example of the use of an arithmetic jump. Assume a byte has been moved into the right half of register 1. The ASCII equivalent of the character "0" is subtracted from it and the result is then tested to see if it is within the range 0 to 9; if not, control goes to the statement (not shown) with the label ERR. (This example is on page 24 of the Dow Editor/Assembler manual.)

```
AI    R1,-48   Subtract "0".
JLT   ERR      Jump if the result is <0.
CI    R1,9     Now compare to 9.
JGT   ERR      Jump if the result is >9.
```

Here is an example of the use of a logical jump. These instructions perform a logical comparison between corresponding characters in two strings. One string is pointed to by register 4, the other by register 5. (This sequence is part of the sort subroutine, XBSORT, in the chapter which shows the sort routine as an example.)

```
CB    *R4+,*R5+   Compare bytes.
JL    OKAY        Go if first less than second.
JNE   SWITCH      All done if not equal.
```

For each byte, if the first is logically less than the second ("lower"), the strings must be in order, so jump to OKAY with JL (jump low). For instance, look at the third bytes of "BAN" and "BAT"; "N" is less than "T", so the program would jump to OKAY.

Suppose on the other hand that "BAT" and "BAN" are being compared. In this case, when comparing the third bytes, the first would not be less than the second. The program next tests to see if they are the same, using JNE (jump not equal). Since in this example the bytes are not the same, the the program jumps to SWITCH. Because the "T" is not less than "N" and not equal to it, it must be greater. Therefore the strings are out of order. If the bytes had continued to be the same up to this point, it would be necessary to check the next byte.

The difference between arithmetic and logical comparisons is simply how the sign bit is treated. The **sign bit** is the left-most bit in a word or byte. Unsigned or logical numbers are never negative. The left-most bit of a word therefore equals exactly 32,768 (or 2 to the 15th power) and the left-most bit of a byte equals 128 (or 2 to the 7th power).

Here are some values shown in hexadecimal first, then with two decimal equivalent values. The first decimal value is the **signed, two's complement**, or **arithmetic** value, while the second is the **unsigned**, or **logical** value.

HEXADECIMAL	SIGNED/ARITHMETIC	UNSIGNED/LOGICAL
0000	0	0
0001	1	1
FFFF	-1	65535
8000	-32768	32768
FF9C	-100	65436

Two jump instructions test to see if there has been a **carry** as the result of an arithmetic (or arithmetic left shift - SLA) operation. JOC jumps on carry, and JNC jumps if there was no carry. Generally, you will want to write programs so that you can avoid carry conditions, but these instructions are available when needed. In particular, the chapter on looping describes how to use JOC for loops.

Another jump instruction, JNO (jump no overflow), tests for the **overflow** condition. This is useful for detecting whether a division actually took place or was aborted because the quotient would not have fit into a single word. Again, you most often will be able to avoid this situation by knowing the range of values your program will be operating on.

Finally, the last jump, JOP (jump odd parity), tests the **parity** of a byte. This is not apt to be used in normal programming. It is very useful when writing highly technical code for doing input and output and checking for data errors.

The **BL** instruction, **branch and link**, is used for subroutine calls. It was described in the chapter on the "first program" and also above during the discussion of B (which is used for the return from the subroutine).

You probably will not use the remaining instructions at first. They are only mentioned here so that you know they exist. There is no real need for you to try to understand them at this time. One of these, **X**, is used to execute a **remote** instruction. This would be useful if you had a list of instructions, one of which is to be executed depending on the value in a register (somewhat like the indexed branch described above). Another, **XOP**, allows you to pretend the computer has an instruction which you define essentially by writing a subroutine.

The last two are used for subroutine calls and subroutine return statements, much like BL and B, and they are in fact used very frequently. The difference between using **BLWP** and **RTWP** as opposed to BL and B is whether or not the subroutine uses the same registers as the calling program. Since the workspace

registers are not hardware registers but simply 32 bytes (16 words) of memory, there can be as many sets of registers as needed or as memory allows. When you call a standard subroutine in ROM, you will use BLWP. This means that the subroutine will not alter your registers but use its own. The subroutine will use RTWP to return to your program. However, when you call a routine which you write, you may prefer to use BL instead for the simple reason that you have defined several of the registers to have global use throughout your program and any subroutines it calls. The use of these instructions will become evident in the example programs later in this book. Bear in mind that a subroutine is written explicity to be called with BL or BLWP and that you cannot use the two instructions interchangeably.

Compare Instructions

By now you should have no trouble understanding the compare instructions. First, most of them look very much like the arithmetic instructions that you should be familiar with by now. Second, they are used in conjunction with the jump instructions, which were just discussed in detail.

There are three instructions, **C**, **CB**, and **CI**, which have exactly the same format as the three addition instructions, **A**, **AB**, and **AI**. The first compares two words, the second compares two bytes, and the third compares a register to an immediate value. Here are examples.

```
C    R1,R12      compare register 1 to register 12.
CB   @H09,R5     compare byte at H09 to first byte.
                 of register 5. (cannot compare to
                 byte of a register.)
CI   R0,>8723    compare register 0 to >8723
                 (first operand must be a register.)
```

Remember that you usually do not need to use a compare instruction to compare something to 0. The reason for this is that many other instructions automatically compare to 0. For instance, after doing A (addition), you could use a jump to see if the result is 0.

There are two other compare instructions that are somewhat different. You can write a lot of assembly language code without having to use them. However, assembly language programs frequently manipulate **bits** (rather than integers or real numbers as in Basic). These two compare instructions enable you to check one or more bits within a word. One is **COC**, compare ones corresponding, and the other is **CZC**, compare zeros corresponding. After executing either instruction, you would use either a JEQ or JNE instruction. (No other jump instructions would be appropriate.) Here is an example.

```
COC R1,R12       See if register 12 has 1's to match
                 each of the 1's in register 1.
```

In order to see how this works, change the little program we used to experiment with the arithmetic instructions as shown on the next page.

Compare Instructions

```
        TITL 'TEST COC INSTRUCTION' (DSK1.TESTA)
        DEF  A
A       MOV  @>A802,R0      LOAD R0 FROM LOCATION A802
        COC  @>A800,R0      COMPARE CONTENTS OF A800 TO R0
        JEQ  EQ             JUMP TO EQ IF BITS ARE 1'S
        CLR  R0             RETURN 0 IF NOT 1'S
        JMP  OUT
EQ      LI   R0,-1          RETURN -1 IF 1'S
OUT     MOV  R0,@>A804
        B    *R11
        END
```

The second value passed from Basic (through location A802) is moved into register 0, then the first value (in A800) is compared to it. If they are equal, the value -1 is returned; otherwise, 0 is returned. Before running the Basic program to call this, change 6 to 4 in statement 210 and delete statement 340.

Here are two examples, first comparing 1 to 3, and then comparing 3 to 1. Remember that the value 3 has two bits and the value 1 has only one bit; and furthermore, that one bit is common to both 3 and 1. That means that each bit in 1 is within the bits in 3, but 3 has a bit not in 1.

First example: Since the value 1 only has one bit on (the far right bit) and the value 3 consists of 2 + 1 (so it has the two rightmost bits on) the condition is clearly equal (which passes back FFFF).

```
(0001)  1
(0003)  3
(FFFF) -1
```

Second example: The value 3 has two bits, only one of which is present in the value 1, so the result is not equal (0000). It is important, of course, not only that the right number of bits are on in the second operand but that the correct bits are on.

```
(0003)  3
(0001)  1
(0000)  0
```

Try this yourself with more complicated examples to get a good feel for this type of operation. Incidentally, you can think of the first operand as "masking out" part of the second. The first operand is a mask or template which determines which bits of the second operand are to be kept.

Transitivity means you can do an operation with either operand in the first position. For instance, Addition is transitive: it doesn't matter whether you add 1 + 3 or 3 + 1. On the other hand, subtraction is not transitive: 1 - 3 is not the

same as 3 - 1. From the examples, you can see that COC is not transitive.

Once you understand the COC instruction, the **CZC** instruction is a simple modification. The format is exactly the same. However, it is not used to test for 1's in the second operand, but for 0's.

Load and Move Instructions

The **MOV** instruction has been used many times already in this book. It simply moves a 16-bit value from one place to another, such as from a memory location to a register. In so doing, the value is compared to 0 (so that you can follow it with a jump instruction if desired). Read "MOV @A,@B" as "move A to B", which would be the same as Basic "B=A".

It is very important to remember that instructions such as MOV which operate on words in memory must always be given an even address. If given an odd address, the low order bit is dropped to make it even. Thus, if you specified >A8F3 for an address, it would be treated as >A8F2. If you make this mistake, your program may behave erratically and you may spend a lot of time trying to locate the problem.

MOVB is exactly the same as MOV, except of course it only moves a single byte. If the operand is a memory location, it can be either even or odd. However, registers are in effect always even memory locations, so that if a register is specified in the move, the byte is moved from (or to) the left side of the register.

You have also seen the **LI** instruction. This is used to assign a specific value, usually to a register. It would have made more sense to have called it MOVI, since LI is to MOV as AI is to A: the value used is right there in the instruction. Read "LI R5,100" as "load register 5 with 100"; this is the same as Basic "R5=100", except that the value on the right of the "=" must always be an integer constant (positive, negative, or zero). Remember that if you want to set a location or register to 0, you may want to use CLR instead because it does not need to store the value 0 in an entire 16-bit word as part of the instruction.

The next most used instruction from this set is probably swap bytes, **SWPB**. This causes the left and right sides of a register or memory location to be exchanged. This is very useful after moving a byte into a register with the intention of performing arithmetic on it, because usually the value should be in the right half of the word for arithmetic.

On the next page is an example, first in Basic and then in assembly language, in which a value is moved from memory location >8375 into register 1 and is then biased by decimal -48. (This location contains the equivalent of the argument K in Basic CALL KEY(unit,K,status). Biasing by 48 in Basic or assembly language changes the character "0" into the value 0, the character "1" into the value 1, and so forth.)

```
100 CALL KEY(0,K,STATUS)
110 PRINT K-48
120 GOTO 100

CLR   R1
MOVB  @>8375,R1
SWPB  R1
AI    R1,-48
```

Another instruction frequently used is **LIMI**, load interrupt mask immediate. Interrupts are discussed in the chapter "Interrupts, Screen, and Keyboard." At this point, it is sufficient to state that this instruction allows you to turn interrupts on or off. You will have to have interrupts turned on in order to produce sounds, and you have to turn them off in order to display anything on the screen. The format of the instruction is LIMI 0 to turn them off, and LIMI 2 to turn them on.

Both **LWPI** and **STWP** are used when you want to control the workspace registers. However, as explained in the chapter on jumps and branches, this control is often not necessary since the BL instruction can be used to call subroutines, leaving the same workspace registers in use.

Finally, **STST** is used to store the status register in a workspace register. Typically, a programmer does not care about the status register directly, but tests some of its bits by using the jump instructions - remember that arithmetic operations and comparisons set various bits in the status register.

Logical Instructions

Before discussing the various instructions, the meaning of the term **logical** must be explained. Rest assured it is not the opposite of "illogical." Rather, logical is a term which refers to operations on a bit by bit basis, rather than on the entire set of bits as just one arithmetic value. Sometimes the term **bit-wise** is used for such instructions.

The first few instructions to be discussed here are really very simple. Each performs exactly the same operation on every bit in a word. The first, **CLR**, for "clear", sets every bit to 0; this instruction has already been used more than once in this book. The second, **SETO**, sets every bit to 1; it means "set to ones." In both cases, all bits take on the same value, regardless of their original values. This is not so with the third instruction, **INV**, which means to invert each bit. This just means that each bit is changed to be the opposite of its original value; each 1 becomes a 0, and each 0 becomes a 1. The technical term of this resulting value is the **one's complement** of the original value. Here are some examples.

```
CLR   R4      Clear register 4.
CLR   *R2     Clear the word pointed to by register 2.
SETO  @FLAG   Set FLAG to hex FFFF (equals -1).
INV   @A(R3)  Invert A(R3).
```

In the first example, no matter what register 4 started out containing, it ends up with all 0's. In the second, whatever 2 locations register 2 points to are both set to 0's. (Note that there is no "clear byte" instruction.) In the third example, the word FLAG is set to -1, regardless of its original value. Finally, in the last example, the value in register 3 is added to the address A to point to two locations (i.e., a word) in memory. The resulting value in that word depends on the starting value. Suppose that A is at location >A000 and register 3 contains 20 decimal, which is 14 in hex. Then the two locations to be A014 and A015. Suppose that word contains hex F35E. Let's look at that in binary to see what the inverse would be.

```
Original value
hex              F    3    5    E
binary       1111 0011 0101 1110

Inverted value
hex              0    C    A    1
binary       0000 1100 1010 0001
```

See how each individual bit is changed. All you need to do to figure out the hexadecimal equivalent value for the entire word is to be able to picture what happens to each digit. Notice that for each digit, the original and the resulting values always add

to 15 (in hex, F). Thus, F+0=F, 3+C=F, 5+A=F, and E+1=F. You can therefore invert a hex digit by subtracting it from 15.

All three of the instructions discussed above operate on only one value. The remaining all operate on two values. Just as with arithmetic instructions, there are both **unary** and **binary** operations. The following instructions are all binary.

In each case, each bit in the result is determined by somehow combining the corresponding bit in the two operands. For example, below are two 16-bit values that were chosen at random. Four of the 16 pairs have been identified with letters below the bits. A 0 or 1 will be placed where each letter is, depending on the values of the two bits immediately above it. The values of any bits to the left or right do not affect this determination.

```
1 0 0 1 0 1 0 0 1 0 1 0 0 1 0 1
0 0 0 1 1 0 1 0 1 1 0 1 1 1 0 0
    a     b           c     d
```

Notice that the four letters represent the four patterns that are possible: 00, 01, 10, and 11. Below is a table which has column headings corresponding to these four letters. A discussion of all the various ways that two bits can be combined will make these specific logical instructions more understandable. The table below shows all possible combinations, with three named in the right margin.

First bit		0	0	1	1	
Second bit		0	1	0	1	
Resulting bit		a	b	c	d	
Combination	1	0	0	0	0	
Combination	2	0	0	0	1	AND
Combination	3	0	0	1	0	
Combination	4	0	0	1	1	
Combination	5	0	1	0	0	
Combination	6	0	1	0	1	
Combination	7	0	1	1	0	EXCLUSIVE OR
Combination	8	0	1	1	1	INCLUSIVE OR
Combination	9	1	0	0	0	
Combination	10	1	0	0	1	
Combination	11	1	0	1	0	
Combination	12	1	0	1	1	
Combination	13	1	1	0	0	
Combination	14	1	1	0	1	
Combination	15	1	1	1	0	
Combination	16	1	1	1	1	

Three of these have been identified because the resulting bit depends on the two operand bits in an interesting and useful fashion.

Logical Instructions

The result labelled **AND** has a 1 only if both of the operand bits have a 1. Incidentally, this is like multiplication of two single bit values, since the result is 1 only if both factors are 1: 0*0=0, 0*1=0, 1*0=0, but 1*1=1. This is also similar to the way we use the word "and" in normal conversation. Example: I use a knife and a fork to eat steak.

The result labelled **INCLUSIVE OR** has a 1 if either or both of the operand bits is a 1. There is no exact analog to this in arithmetic, although you would get the same result if you added the two bits and then added the carry bit, if any, back into the sum. That is, 0+0=0, 0+1=1, 1+0=1, and 1+1=10=1. Often the word "or" when used in plain speech has the same meaning. For example: I could swim or ride a bicycle to get exercise (either or both is possible). Sometimes people use "and/or" for this.

The result labelled **EXCLUSIVE OR** has a 1 if only one of the operand bits is a 1, but if both are 1's the result is 0. This same result can be obtained by adding the two bits and discarding any carry bit. Thus, 0+0=0, 0+1=1, 1+0=1, but 1+1=10=0. Sometimes when we use the word "or" it has this sense, as in: you have to pull on your left pant leg first or your right pant leg first (but you can't do both at the same time).

The other results are not seen as often either as instructions on computers or for that matter in computer languages. Some are omitted because they do not depend at all on the operand bits: see the combinations 1 and 16. Others are not symmetrical, so it matters which operand has the 1 or 0: thus, combinations 3 and 5 are identical, except for the order of the operands.

The description above should help you understand what happens with each bit. If you remember that the exact same operation is applied to each of the bits in the word (or byte if it is a byte instruction), you should understand the instruction. Incidentally, in the chapter on comparisons, the instructions **COC** and **CZC** also worked in a **bit-wise** fashion. They differ from logical instructions in that they do not generate a result that is stored.

Logical Instructions

Here are examples of bit-wise, or logical operations, applied to entire words. Each is shown in binary and hex notation.

```
Example of AND
First word          1001 0101 0100 1011  954B
Second word         1100 1101 0101 1010  CD5A
Result              1000 0101 0100 1010  854A

Example of INCLUSIVE OR
First word          1001 0101 0100 1011  954B
Second word         1100 1101 0101 1010  CD5A
Result              1101 1101 0101 1011  DD5B

Example of EXCLUSIVE OR
First word          1001 0101 0100 1011  954B
Second word         1100 1101 0101 1010  CD5A
Result              0101 1000 0001 0001  5811
```

The various logical operations have now been discussed. Here is the description of the instructions that are used for these operations.

To **AND**, you can use "and immediate", **ANDI**. This has the same format as "add immediate". **SZC**, set zeros corresponding, is another instruction which performs a modified "and." It is combination 5 instead of combination 2, so it does the same as performing an inversion on the first operand and then anding the two. It has the same format as the addition instruction, A. This is an example of using these two instructions.

```
ANDI  R12,>F00F    And register 12 by hex F00F.
SZC   @M,R2        Set register 2 to the result of
                   SZCing M and register 2.
```

Here is an example of how the result is computed for SZC.

```
Example of SZC
First word          1001 0101 0100 1011  954B
After inversion     0110 1010 1011 0100  6AB4
Second word         1100 1101 0101 1010  CD5A
Result              0100 1000 0001 0000  4810
(based on first word
after inversion)
```

There is also a **SZCB** instruction, which operates on bytes. Remember that if the operand is a register, only the left byte is used.

To do an **INCLUSIVE OR**, there is **ORI**, "or immediate", and **SOC**, "set ones corresponding." The ORI has the same format as ANDI and SOC has the same format as SZC. Note that SOC actually performs an "or", unlike SZC which performs a modified

"and". There is also **SOCB**, which performs an "or" on bytes: it has the same format as SZCB.

To do an **EXCLUSIVE OR**, there is only the **XOR** instruction. The result is stored in the second operand which must be a register. Here is an example.

```
        XOR   @ABC,R3   Exclusive or ABC to register 3.
```

You can modify the little program used to test arithmetic instructions to become familiar with these various operations. Here it is with the SOC instruction.

```
        TITL 'TEST SOC INSTRUCTION' (DSK1.TESTA)
        DEF  A
A       MOV  @>A802,R0      LOAD R0 FROM LOCATION A802
        SOC  @>A800,R0      SET ONES, LOCATION A800 TO R0
        MOV  R0,@>A804
        B    *R11
        END
```

This is what happens when the Basic program calls this with the values 1 and 2, and then -256 and 10.

```
(0001)  1
(0002)  2
(0003)  3

(FF00) -256
(000A)  10
(FF0A) -246
```

You should of course try many other examples on your own until you develop a good feel for how the instruction works. It will also help make you familiar with hexadecimal notation.

Try the assembly language program with SZC and XOR also.

Before ending the discussion of logical operations, it is necessary to put them into the proper perspective. Usually these instructions are of value in manipulating data within bytes or words. For instance, you may want to create the bit pattern to make a particular shape to be displayed on the screen. Another example would be plotting with a dot-matrix printer. It is helpful to think of the "and" operation as masking: you can use it to "knock out" or "strip away" some of the bits in a word or byte. On the other hand, the "or" is used to meld two values together into one again. Frequently an "and" operation will use a mask such as hexadecimal FF00 (to knock out the right half) or 00FF (to knock out the left half). And frequently an "or" will have a value only having "1" bits in the left half (such as hex 0C00) and another value only having "1" bits in the right half (such as

hex 0068).

Incidentally, these instructions set the status register so that you can follow with a jump instruction to test the result against 0.

Finally, if you use Extended Basic, you may not be aware that you can perform AND, OR, XOR, and NOT (inversion) in Extended Basic. The following program listing demonstrates the use of AND. (See statement 140.)

```
100 REM TEST LOGICAL OPERATIONS IN EXTENDED BASIC
110 FOR I=1 TO 2
120 INPUT N(I)
130 NEXT I
140 N(3)=N(1)AND N(2)
150 FOR I=1 TO 3
160 T=N(I):: IF T<0 THEN T=T+65536
170 BYTE1=INT(T/256)
180 BYTE2=T-256 * BYTE1
190 IF N(I)>65535 THEN N(I)=N(I)-65536
200 PRINT N(I);
210 B=BYTE1
220 GOSUB 280
230 B=BYTE2
240 GOSUB 280
250 PRINT ") ";N(I)
260 NEXT I
270 GOTO 110
280 REM BINARY TO HEX CONVERSION
290 HEXDIG1=INT(B/16)
300 HEXDIG2=B-16*HEXDIG1
310 PRINT SEG$("0123456789ABCDEF",HEXDIG1+1,1);
SEG$("0123456789ABCDEF",HEXDIG2+1 ,1);
320 RETURN
```

Shift Instructions

After all the experience you have just gained with logical operations, it should now be quite easy to pick up an understanding of the shift instructions. These are in fact basically logical instructions, in that they are also used to manipulate bits within words.

In order to understand these instructions, you must think of word values as strings of bits. There are two fundamental types of shifts - shift left, or shift right.

Here is an example of shifting a string of 16 bits to the left. Each line of the example represents shifting another bit. It is shifted a total of 5 times. The vertical lines represent the word boundaries. That is, values outside the boundaries do not exist anymore.

Original value		0110101010011101
After shifting once	0	1101010100111010
After shifting twice	01	1010101001110100
After shifting three times	011	0101010011101000
After shifting four times	0110	1010100111010000
After shifting five times	01101	0101001110100000

There is a difference between shifting to the right and shifting to the left, as in the example above. In the shift to the left, 0's were filled on the right as the bits already there moved across to the left and vacated their positions. In the shift to the right, it is possible to fill with 0's on the left. However, there is also the option of filling with either 0's or 1's, depending on what was initially present in the left-most bit. (Incidentally, the bit on the left is the **sign-bit** and is numbered 0, and the bit on the far right is numbered 15.) Here is what happens with a right shift that extends the sign-bit.

Original value	1010101010011101	
After shifting once	1101010101001110	1
After shifting twice	1110101010100111	01
After shifting three times	1111010101010011	101
After shifting four times	1111101010101001	1101
After shifting five times	1111110101010100	11101

The other possibility for filling on the left side is to pick up the bit that was dropped off the right side. It would also be possible to fill on the right with the bit that was dropped off the left side. Either is called a **circular shift.** A computer designer could make a circular shift go in either direction. However, both are not really necessary, in that a circular shift to the right by 1 bit is the same as a circular shift to

the left 15 bits. Here is an example of circular shifting to the right.

Original value	1010101010010101
After shifting once	1101010101001010
After shifting twice	0110101010100101
After shifting three times	1011010101010010
After shifting four times	0101101010101001
After shifting five times	1010110101010100

Having looked at all these examples, it is now appropriate to list the instructions actually available on the TI.

SRC	Shift right circular
SRA	Shift right arithmetic (fills on left with sign bit)
SRL	Shift right logical (fills on left with 0's)
SLA	Shift left arithmetic (fills on right with 0's)

For all of these, the first operand is a register that contains the value to be shifted. You must also specify how many bits to shift with the second operand, and you use the same method for all of them. You can either specify the shift count in the instruction itself, or you can specify 0 in the instruction, which means to use the value in register 0 to specify how many to shift.

Here are two examples. Notice that the first shifts the contents of the register by 3, which is specified in the instruction, and so it is very similar to immediate addressing.

```
SRA   R6,3    Shift right arithmetic register 6 by
              3 bits.
```

In the second, the number of bits to shift is not in the instruction itself but is specified in register 0.

```
SRL   R3,0    Shift right logical register 3 by
              the value in register 0.
```

Here is an assembly language example of the SRC instruction. You should try it as well as the others.

```
      TITL 'TEST SRC INSTRUCTION' (DSK1.TESTA)
      DEF  A
A     MOV  @>A802,R1     LOAD R1 FROM LOCATION A802
      MOV  @>A800,R0     LOAD R0 FROM LOCATION A800
      SRC  R1,0          SHIFT R1 BY CONTENTS OF R0
      MOV  R1,@>A804     STORE RESULT IN A804
      B    *R11
      END
```

Here are several sample results of calling this program, using the same small Basic program (DSK1.TESTB) used in previous chapters.

```
(0001)  1
(0002)  2
(4000)  16384

(FF00) -256
(0004)  4
(0FF0)  4080

(0003)  3
(0001)  1
(8001) -32767
```

In the first example, the value 1 is shifted out of the right position (bit 15) into the left-most bit and then again into the second bit, making hex 4000. In the second example, 8 bits can be seen to shift right 4 bits (which is one hex digit), thus changing FF00 to 0FF0. In the third example, two 1's are shifted right just 1 bit, leaving one on the right side and moving one to the left side - this yields hex 8001.

Here is a more elaborate example, combining a shift and a loop structure. The purpose of this program is to find the highest power of 2 in a given number. Due to the way it is written, you cannot give it a number greater than 16,384. To use the Basic test program (DSK1.TESTB) to call it, you may wish to change statement 120 to be "I=0", delete statements 190 and 340, and change "6" to "2" in statement 210.

```
        TITL 'FIND HIGHEST POWER OF 2'
        DEF  A
A       MOV  @>A800,R0      LOAD R0 WITH VALUE
        LI   R1,1           SET R1 = 1
L       C    R1,R0          IS R1=R0 OR R1>R0?
        JEQ  OK             R1=R0.  FOUND LARGEST VALUE.  QUIT.
        JGT  DEC            R1>R0.  TOO LARGE.  QUIT
        SLA  R1,1           MULTIPLY R1 BY 2
        JMP  L              GO BACK TO TRY AGAIN.
DEC     SRA  R1,1           TOO BIG, SO DIVIDE BY 2.
OK      MOV  R1,@>A802      RETURN RESULT IN A802
        B    *R11
        END
```

The program puts the value you specify into register 0. The value 1 is put into register 1. Then the 1 is shifted repeatedly to the left by 1 until it is equal to or greater than the value you entered. If it is greater than your value, it is shifted back to the right once.

Shift Instructions

Here are the results of entering 1 and 19.

```
(0001)  1
(0001)  1

(0013)  19
(0010)  16
```

Directives

Commands that you give to an assembler are called "directives". None of them generate any instructions for the computer to execute. Instead, they are used to control the listing, to make the program load at a particular location, to reserve space in memory for data, to link to other programs, and so forth.

Some directives are common to both TI's Editor/Assembler and the Dow Editor/Assembler, and some are not. All are discussed here, and the differences between the two assemblers are described.

When using the TI Editor/Assembler, you must put the **END** directive at the end of each program. This is not necessary with the Dow Editor/Assembler, since the editor and assembler functions are combined in one program, which knows where the end of your program is. There is nothing complicated about this directive: it just marks the end.

If you are using the TI Editor/Assembler, you have access to a very powerful feature known as "linking". By means of the **REF** and **DEF** directives, the loader is able to link together the various parts of your program and link your program to library routines (that is, utilities). What this means in practical terms is that your program can refer to something by its name instead of by its location in memory. (By comparison, using the Extended Basic loader or the Dow Editor/Assembler, you must know the location and enter it into the program with an EQU statement.)

If you want a program to be callable by another program, you use the DEF statement to list any labels which are to be known "externally". (Naturally any label in your program can be used from within the program itself: DEF makes a label available outside your program.) If your program refers to something outside itself, it must list that label on a REF statement. Here are examples.

```
        DEF   MYSUB
        REF   SUB1,SUB2
MYSUB   ...   (The program called MYSUB starts here)
        ...
        BL    @SUB1   (This is a call to SUB1)
        ...
        BL    @SUB2   (This is a call to SUB2)
        ...
        END
```

This represents the fragments of a program which can be called by the name MYSUB and which in turn calls SUB1 and SUB2.

The DEF statement declares that MYSUB can be used outside this program, presumably by some other program that you write. The REF statement declares that SUB1 and SUB2 are defined in another program, perhaps a subroutine you wrote earlier.

Use **DEF** to define the entry point for a program so that it can be called by name to be run. For instance, the name you specify with DEF is the name you refer to in the CALL LINK in Basic or Extended Basic.

EQU is the directive used instead of REF when loading assembly language programs with Extended Basic or when using the Dow Editor/Assembler. It has other uses as well because what it actually does is assign a value to a symbol in the program. Whenever that symbol is used in the program, it is as if the value itself had been used. For instance, here is an example that was taken from the Tombstone City game included as an example with the TI Editor/Assembler. It illustrates the use of EQU to give a meaningful name (SHIPRT) to a hexadecimal value (>68).

```
SHIPRT EQU  >6800
       ...
       LI   R4,SHIPRT
       MOVB R4,@SHIP
```

This assembles into exactly the same machine language instructions as if these two statements had been used.

```
       LI   R4,>6800
       MOVB R4,@SHIP
```

In both cases, the byte value >68 is moved into location SHIP. In this instance, >68 is apparently a character pattern instead of an address. As another example, you may want to use a symbol to indicate a constant which is used in several places in your program. You could use the symbol NSTARS to mean the number of sprites that look like stars zooming around on the screen. The symbol would of course only be defined once, like this: NSTARS EQU 20. Then, if it becomes necessary to change the value, you need only change the value in the EQU statement.

Do not confuse the assembly language statement "A EQU 5" with the Basic statement "A = 5". Using the EQU statement to give a symbol a value is a concept entirely alien to Basic. Whether in Basic or assembly language, a symbol such as "A" has a value which is the address at which the contents of the variable is stored. The difference is that in Basic you have no need to know what the address is, and so Basic programmers do not even need to know that "A" is a symbol with a value.

An interpreted language such as Basic maintains a symbol

table. The symbol "A" would be an entry in the table, and stored in the table would be its value, the address. Each time the variable "A" is used, the interpreter finds the symbol "A" in the table, and then uses the address stored with it to retrieve or store the value of the variable (5 in the example in the preceding paragraph).

In assembly language you may also not know the actual address, since a symbol can be assigned a value by usage as a label (and in other ways to be discussed below). However, if you desire, you can use a directive such as EQU to force a symbol to have a known value.

There are three ways of giving a symbol a value: use it as a label, use REF (if using the TI Editor/Assembler loader), or use EQU. When using EQU, usually a symbol will be assigned an address represented as a hexadecimal constant. It could be the address of a utility routine or it could be the address of your own routine. For instance, if using the Dow Editor/Assembler there is a limit to program size, so different parts have to be assembled and loaded individually: you should then use EQU's to enable each part to refer to the other parts. Here is a simple example of an EQU, although the examples that follow later in this book will be much more meaningful.

```
STRREF EQU  >2014     (For Extended Basic)
STRREF EQU  >604C     (For Mini Memory Module)
```

Having included the EQU for STRREF in your program, you can now branch to STRREF with BLWP. (STRREF is a utility that copies a character string from a Basic program into your assembly language subroutine.)

Next in order of usefulness after the directives described above is probably the **DATA** directive. This allows you to put data into your program. Usually you will have a label on it as well. It is not the same as the DATA statement in Basic, since in Basic you must use a READ statement to use any value in a DATA statement, while in assembly language you address the value directly by the label.

An example of using the DATA directive is if you want to divide something by 100. There is no "divide immediate" instruction which would allow you to specify the value 100 right in the instruction. Instead, you can put the value 100 in a word in memory, and then refer to that location when doing the division, like this...

```
        DIV   @V100,R5     Divide 100 into register 5
        ...
V100    DATA  100          The value 100.
```

(Notice that the label used here suggests the value itself: "V100" for "value 100".) The value you assign in a data statement can be a decimal or hexadecimal constant. With TI's Editor/Assembler, but not with the Dow Editor/Assembler, it can also be a string.

The **BYTE** directive is very similar to DATA, except that it loads individual bytes, not whole words. This directive is useful for byte oriented instructions. Remember to watch out for creating odd addresses in your program by defining an odd number of bytes. Many programmers always insert an EVEN directive after one or more BYTE directives.

After using BYTE with an odd number of bytes, the location counter for the TI Editor/Assembler may have an odd value. However, if you follow it with an instruction or DATA directive, the location will automatically be incremented to the next even value, since values addressed by word oriented instructions and instructions themselves must be loaded at even addreses. If necessary, you can use the **EVEN** directive to force the location to be even. (The Dow Editor/Assembler automatically prevents the odd address problem from happening with the BYTE directive. It differs from the TI Editor/Assembler also in that decimal values must be positive.)

With either the TI or Dow Editor/Assembler, you can define a string of characters without having to use BYTE to define each character individually. **TEXT** is very similar to DATA and BYTE in that it stores the values you specify and allows you to have a label for them. A difference is that it is restricted to characters that you can enter with the editor. This is what it looks like:

```
SAYBYE TEXT  'Bye now, see you later'
```

(The Dow Editor/Assembler allows you to use any break character to define the string in the TEXT statement. In the example above, the quotation mark was used for the break character. You could, for instance, also use the slash (/) if you wish.)

The Dow Editor/Assembler also differs by having an additional directive, **BTXT**. This stands for "Basic TEXT" and is used when you want to define a string to be displayed on the screen by a subroutine called from a Basic program. The reason for this extra directive is rather obscure, having to do with the way Basic uses VDP memory as efficiently as possible. What it amounts to is a bias of hexadecimal 60 added to each character before being stored in the screen image in VDP memory. This bias amount is automatically added by BTXT so that characters are displayed correctly. If you are writing a program to be run independently of Basic, be sure to just use TEXT. Here is a sample assembly language program to display data on the screen. (The

program is very simple. All it does is call a utility program to copy data to the screen. This will be discussed later in the Primer.)

```
TEST TEXT AND BTXT DIRECTIVES
000     CLR  R0         LOAD R0 WITH VDP ADDRESS
002     LI   R1;TXT     LOAD R1 WITH CPU ADDRESS
006     LI   R2;5       LOAD R2 WITH CHARACTER COUNT
00A     BLWP @MBW       SEND DATA FROM CPU TO VDP
00E     B    *R11
010 TXT:TEXT /HELLO/    TEXT TO BE SENT (WRITTEN ON
                        SCREEN)
016 MBW:EQU  >6028
```

Load the assembly language program into the Mini Memory Module, then run it by selecting the Mini Memory option and then selecting "RUN". Type the same name for the program that you loaded it into the REF/DEF table. Watch closely - the word HELLO will flash rapidly in the upper corner of the screen.

Now enter this Basic program and run it.

```
100 CALL CLEAR
110 CALL LINK("TEMP")
120 GOTO 120
```

This time the screen is blank because the TEXT directive did not bias the characters as the BTXT directive would. Now go back to the assembly language program and change TEXT to BTXT at location 010. Again run the Basic program. Now it should say HELLO in the corner of the screen.

If you want to define an area of memory to be used as a list of values or as a string, the best way is to assign a label to it and reserve space with the **BSS** directive. This means "block starting with symbol".

This statement will reserve 100 bytes, which is 50 words, and give it the label BUF.

```
BUF    BSS    100      Buffer space.
```

BSS is somewhat similar to DIM in Basic. However, there are some very significant differences. This is what a DIM statement in Basic looks like.

```
DIM N(25),S$(15)
```

First, BSS allocates memory exactly where it occurs in your program, whereas you the programmer have no idea where Basic has obtained space for your variables. Second, BSS allocates exactly as many bytes as you specify, whereas Basic allocates eight bytes

for each numeric value and as many as are necessary for each string in your array. Thus, eight bytes are allocated for each of N(1), N(2), N(3), and so forth. And a variable number of bytes is allocated to each of S$(1), S$(2), S$(3), and so forth. Remember that in assembly language, the address of the first location assigned to the label (BUF in the BSS example above) is equated to the symbol. Therefore, if you are computing an index into the space, don't forget that the first word is indexed by 0. Also, remember that you must increment the index value by 2 for each new word value.

The TI Editor/Assembler also has the directive **BES**, which is "blocked ended by symbol". This is useful if for some reason you want to use negative index values, since they would be subtracted from the label at the end of the space.

When you load a program with the TI Editor/Assembler or Extended Basic, you should not be concerned about its location in memory. The loader worries about that. If your program assigns all the memory space it needs with BSS (or perhaps BES), you should have no problem. However, it is possible to control where your program is loaded by using the **AORG** directive. This allows you to specify an absolute address, and that part of your program which follows the directive will start loading at the specified location. This should only be used when absolutely necessary and with extreme caution, since you could cause the program to load on top of something else.

With the Dow Editor/Assembler there is no AORG directive because every time you load you use the **LOAD** command and specify the absolute address where the program is to be loaded. The LOAD command is thus very much like the AORG directive. Another difference is that if using the Dow Editor/Assembler, you may wish to assign blocks of space using EQU rather than BSS. The reason for this is that there is only a certain amount of memory available at one time for the editor, and using it for a block of memory could result in more fragmentation of your program than would be nice. However, if you keep track of how your are using memory with a memory map (in your notebook!), you can use EQU in the program to make a label refer to the appropriate place in memory.

Calling From Basic Programs

There are four routines which can be used to pass data back and forth between a Basic program and an assembly language program. These routines are usually preferable to the technique frequently used so far in this book, which is to have the Basic program call LOAD or PEEK to pass values through specific memory locations.

The four routines are: STRREF and NUMREF (to pass strings and numbers from Basic to assembly language), and STRASG and NUMASG (to pass strings and numbers from assembly language to Basic). They are discussed in the TI Editor/Assembler manual, pages 284 through 287, and in the Mini Memory Module manual, pages 52 through 54.

If you are using Basic with the TI Editor/Assembler Module, just use the REF directive to refer to these routines and be sure to load the file DSK1.BSCSUP. For use with either Extended Basic or the Mini Memory Module, use the following addresses with the EQU directive.

	Extended Basic	Mini Memory Module
NUMASG	>2008	>6040
NUMREF	>200C	>6044
STRASG	>2010	>6048
STRREF	>2014	>604C

For all four routines, registers 0 and 1 must be set with appropriate values before the routine is called. If the argument being passed is not an array element, register 0 should be cleared (set to 0). Register 1 should be set to indicate which argument in the CALL LINK is to be passed. In determining what number to use, do not count the name of the subroutine itself; thus, in CALL LINK("NTOS",A,A$), A is 1 and A$ is 2.

For the two numeric routines, the value is passed to the assembly language program using FAC, the floating point accumulator. It has 8 bytes, starting at location >834A. To refer to it, use FAC EQU >834A.

For the two string routines, register 2 must contain the address of the string in the assembly language program. The first byte of the string indicates the length of the string. For STRREF, the first byte should be set to indicate the longest string that can be accepted from the Basic program. (If the string actually passed is too long, you will get an error message indicating string truncation.) For STRASG, the first byte should indicate the exact string length to be sent to the Basic program.

Calling From Basic Programs

This chapter includes the listing for a program which uses all four calls. The routine STON (for String TO Number) accepts a string as the first argument and just passes it back as a number in the second argument. (See statement 340 in the Basic program below.) The routine NTOS (for Number TO String) accepts a number as the first argument and passes it back as a string to the second argument. (See statement 270 below.)

Since strings and numbers are the two different types of data provided by Basic, these routines perform the **type transfer** function. That is, you can treat a number as a string and vice versa. (This is not the same as VAL and STR$, which convert a number to a string, and vice versa.)

Here is the Basic program which performs the calls. The program has two numeric variables, A and B. Corresponding to each is a string, A$ and B$. The program allows you to enter a number or a string into A or A$, and it then makes the other agree. Each number is stored in 8 bytes, and these can be copied into a string of length 8. For example, if you enter the numeric value 100 into A, the program sets A$ to an 8 byte string having the same contents byte by byte as A. The program has an option to enable you to copy A into B and A$ into B$. It displays both A and B as numbers, then displays the ASCII equivalents for the 8 bytes of A$ and B$. Finally, it performs comparisons between A and B and between A$ and B$ and displays the results of the comparisons.

```
100 REM PROGRAM TO TEST NTOS AND STON
110 CALL INIT
120 CALL LOAD("DSK1.STONOBJ")
130 B=0
140 CALL LINK("NTOS",B,B$)
150 PRINT : : :
160 PRINT "ENTER 1 TO MOVE A TO B"
170 PRINT "ENTER 2 TO ENTER A NUMBER"
180 PRINT "ENTER 3 TO ENTER A STRING"
190 INPUT CHOICE
200 IF CHOICE<1 THEN 150
210 IF CHOICE>3 THEN 150
220 ON CHOICE GOTO 230,260,290
230 B=A
240 B$=A$
250 GOTO 150
260 INPUT "ENTER NUMBER:":A
270 CALL LINK("NTOS",A,A$)
280 GOTO 350
290 A$=""
300 FOR C=1 TO 8
310 INPUT "ENTER BYTE "&STR$(C)&":":BYTE
320 A$=A$&CHR$(BYTE)
330 NEXT C
```

```
340 CALL LINK("STON",A$,A)
350 PRINT : :"A=";A
360 N$=A$
370 GOSUB 540
380 PRINT "B=";B
390 N$=B$
400 GOSUB 540
410 IF A>=B THEN 430
420 PRINT "A<B"
430 IF A<>B THEN 450
440 PRINT "A=B"
450 IF A<=B THEN 470
460 PRINT "A>B"
470 IF A$>=B$ THEN 490
480 PRINT "A$<B$"
490 IF A$<>B$ THEN 510
500 PRINT "A$=B$"
510 IF A$<=B$ THEN 530
520 PRINT "A$>B$"
530 GOTO 150
540 REM DISPLAY BYTES IN STRING
550 FOR C=1 TO 8
560 PRINT ASC(SEG$(N$,C,1));
570 NEXT C
580 PRINT
590 RETURN
600 END
```

The assembly language program (Extended Basic version) is listed on the next page. The numbers in the left margin are mentioned in the accompanying text.

The routine STON must first prepare the space S (at 20 on the listing) to accept the 8 bytes of the string. The first byte has to have an 8, meaning that at most 8 bytes can be received. At 1, the MOVB @DB8,@S puts the value 8 into the first byte of S. (DB8 means "decimal, byte, value 8"; this is just a convenient means of naming the value. The value is defined at 21 on the listing.) At 2, register 0 is cleared (because the value to be received is not an element of an array). At 3, register 1 is set to 1 to get the first argument. At 4, register 2 is loaded with the address of S. Finally, at 5, BLWP is used to call STRREF to move the string.

```
            TITL 'STON AND NTOS'
    *
    *       CALL LINK("STON",S$,N)
    *       CALL LINK("NTOS",N,S$)
    *
    *       To be called from Extended Basic.
    *
    FAC     EQU  >834A
    NUMASG  EQU  >2008
    NUMREF  EQU  >200C
    STRASG  EQU  >2010
    STRREF  EQU  >2014
            DEF  STON,NTOS
1   STON    MOVB @DB8,@S        Move 8 to S (max number of chars)
2           CLR  R0             Call STRREF to move string in first
3           LI   R1,1           argument into S.
4           LI   R2,S
5           BLWP @STRREF
6           LI   R0,S+1         Now copy string from S to FAC.
7           LI   R1,FAC
8           MOVB *R0+,*R1+
9           MOVB *R0+,*R1+
10          MOVB *R0+,*R1+
11          MOVB *R0+,*R1+
12          MOVB *R0+,*R1+
13          MOVB *R0+,*R1+
14          MOVB *R0+,*R1+
15          MOVB *R0+,*R1+
16          CLR  R0             Call NUMASG to pass FAC back to
17          LI   R1,2           second argument.
18          BLWP @NUMASG
19          RT                  All done. Return to Basic program.
    *
    NTOS    CLR  R0             Call NUMREF to move number in first
            LI   R1,1           argument into FAC.
            BLWP @NUMREF
            MOVB @DB8,@S        Move 8 into S (string length).
            LI   R0,FAC         Now copy string from FAC to S.
            LI   R1,S+1
            MOVB *R0+,*R1+
            MOVB *R0+,*R1+
            MOVB *R0+,*R1+
            MOVB *R0+,*R1+
            MOVB *R0+,*R1+
            MOVB *R0+,*R1+
            MOVB *R0+,*R1+
            MOVB *R0+,*R1+
            CLR  R0             Call STRASG to pass string to
            LI   R1,2           second argument.
            LI   R2,S
            BLWP @STRASG
            RT                  All done. Return to Basic program.
```

```
   *
20 S       BSS  9            Space large enough for length byte
   *                         and 8 characters.
21 DB8     BYTE 8            Max string length for NAME.
           END
```

Now that the 8 bytes have been accepted into S, they have to be copied into FAC to be sent back via the second argument. To copy the 8 bytes, the MOVB *R0+,*R1+ instruction is used 8 times, at 8 through 15. Each time it is executed, it moves one byte and increments both registers 0 and 1 to point to the next byte in sequence to be moved. Before the first use, however, it is necessary to make register 0 point to the first byte of the string in S (this is done at 6) and to make register 1 point to the first byte where the string is to be moved in FAC (this is done at 7).

Because the first byte of S contains the length, the first data byte is at S+1. Notice, at 6, the statement LI R0,S+1, which puts the address of the second byte in S into register 0. If you are using the Dow Editor/Assembler, you cannot use S+1. In that case, you have to load register 0 with the address of S (use LI R0,S) and then increment the address with INC R0.

Passing the number back to Basic once it has been moved into FAC is done with NUMASG. At 16, register 0 should be 0 because it is not being passed an array element. At 17, register 1 is loaded with 2 to pass the data to the second argument of the CALL LINK. After calling NUMASG at 18, the program returns to the calling Basic program with RT at 19.

The routine NTOS is nearly identical to STON, so it is not discussed here in detail.

These programs can now be used not only to learn how to pass data but also to learn how numbers are actually stored in the TI. This is described in the TI Editor/Assembler manual on page 279 and in the Mini Memory Module manual on page 27. Here is how to interpret values that are positive, within the range 0 to 9999.

```
If first byte is 64,
     Value = second byte.
If first byte is 65,
     Value = second byte times 100
             plus third byte.
```

Here are some examples of values for A and the 8 bytes in the resulting string.

A	String 1	2	3	4	5	6	7	8
0	0	0	?	?	?	?	?	?
1	64	1	0	0	0	0	0	0
-1	191	255	0	0	0	0	0	0
100	65	1	0	0	0	0	0	0
9999	65	99	99	0	0	0	0	0
32767	66	3	27	67	0	0	0	0

In the first example, A is 0. In that case, only the first two bytes have any meaning - both are 0. (The question marks mean that it does not matter what is in the last 6 bytes.) In the next example, A has the value 1. The first byte, 64, is the **exponent**, which is always biased by 64. Therefore, an exponent value of 64 means 100 raised to the 0 power. The next example, A = -1, has an exponent of 191 and a first byte of 255. Taken together, these values form the two's complement of 64 and 1. (In hex, 64 and 1 is >4001, while 191 and 255 is >BFFF.) When the value of A is 100 or greater, the exponent is no longer 64 but increases to 65 or higher. In all cases, subtract 64 from the exponent to find the correct power. For instance, 65 means multiply by 100, 66 means multiply by 10000, and so forth.

Now use the option to assign a string to A. If you assign the random string 231, 23, 49, 0, 128, 23, 3, 67, to A, it is displayed as -G3.49013E-80, which is nonsense. (The E-80 is the exponent, but the -G3 is garbage.) If you set A and B to the two values below, the number A is greater than the number B, but the string A$ is less than the string B$, certainly a strange result.

```
A = 127 127 127 127 127 127 127 128
B = 127 127 127 127 127 127 127 127
```

Further experimentation will show that numeric comparisons do not work well if you load numbers with strange values. Also, string comparisons treat each byte as a signed value. Therefore, any decimal value between 128 (hex >80) and 255 is less than 127 (hex >7F) because 128 turns on the sign bit for the byte. If you want to test two strings to see which is less, and if you have loaded the strings with values in the range 0 to 255 (instead of just 0 to 127), you must compare individual bytes with SEG$ rather than comparing the strings as a whole.

The Basic program shown above assumes Extended Basic is being used. For Basic with the TI Editor/Assembler, include DSK1. BSCUP in the CALL LOAD on statement 120. For the Dow Editor/ Assembler, delete statements 110 and 120, and follow previous examples in this book to convert the assembly language routine.

A Case History

In this chapter, the actual case history of the development of an assembly language routine will be shown. It will mean that there will be several listings, as the routine starts out simple and develops until it does all that we want. This should give you a better idea of the types of problems that can be encountered when programming in assembly language.

The goal will be to write an assembly language program which will be useful. A way to do that is to enhance what can be done with Basic, and one way to do that is to do something faster. Look at this small program.

```
100 REM TEST MULTIPLE CONCATENATION
130 READ A$,B$,C$,D$,G$,M$,W$,X$,Y$,Z$
140 DATA A,B,C,DEF,GHIJKL,MNOPQRSTUV,W,X,Y,Z
150 INPUT N
160 FOR I=1 TO N
170 MC$=A$&B$&C$&D$&G$&M$&W$&X$&Y$&Z$
180 NEXT I
190 PRINT MC$
200 END
```

When you run it, it will prompt for N. It then goes through the loop N times, allowing you to time how long it takes to concatenate all the strings to assign the result to MC$. When it is done, it prints MC$ so you see that it worked properly. Given the value of 100, I found it to take 22 seconds. This seems rather slow. Just for comparison, if statement 170 is changed to this...

```
170 MC$="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

it only takes 1 second to iterate 100 times. Clearly doing many concatenations (the "&" operation in Basic) might be something that could be improved with assembly language. Here is another version of the program above. It is written with the assumption that there exists a routine named "MC" (for "multiple concatenation") that is being called from Basic using the TI Editor/Assembler. (Modifications for use with the Dow Editor/Assembler and Extended Basic will be shown later.)

A Case History

```
100 REM TEST MULTIPLE CONCATENATION
110 CALL INIT
120 CALL LOAD("DSK1.BSCSUP","DSK1.MCEAOBJ")
130 READ A$,B$,C$,D$,G$,M$,W$,X$,Y$,Z$
140 DATA A,B,C,DEF,GHIJKL,MNOPQRSTUV,W,X,Y,Z
150 INPUT N
160 FOR I=1 TO N
170 CALL LINK("MC",MC$,A$,B$,C$,D$,G$,M$,W$,X$,Y$,Z$)
180 NEXT I
190 PRINT MC$
200 END
```

At this point we have defined a problem and have written an example program showing what we would like to do. Now all that is left is to write the actual assembly language routine to do the multiple concatenation! We can only hope that it will result in an improvement in speed.

Let's start by trying to write MC so that if called with a string as one argument it will return it as another. Here is the test program.

```
100 REM  TEST MULTIPLE CONCATENATION
110 CALL INIT
120 CALL LOAD("DSK1.BSCSUP","DSK1.MCEAOBJ")
130 CALL LINK("MC",MC$,"TEST")
140 PRINT MC$
150 END
```

We want this to copy the string "TEST" into MC$. For the program to work, it is necessary for it to be able to get data from the Basic program and to return data to the Basic program. One reason I have chosen this particular program to demonstrate parameter passing between Basic and assembly language is that strings are very simple. They consist of one or more bytes, usually designating characters. Unlike numbers, strings are thus essentially the same in Basic as in assembly language.

The utilities **STRREF** and **STRASG** can be used from assembly language to get strings from Basic and to return strings to Basic, respectively. It is not necessary to use them, but they make programs much easier to read and do not add enough to the overhead of the program to be a consideration.

In both cases, **register 0** must be set to 0 before calling the utility to indicate that the string to be passed is not an element in an array. **Register 1** must be set to indicate which parameter is to be used - this is a number from 1 to 15. And **register 2** is loaded with the address of the buffer which contains the string in the assembly language program.

A Case History

Here is the initial program. It is the file "DSK1.MCEA".

```
         TITL 'MC - Multiple Concatenate'
         DEF  MC
         REF  STRREF,STRASG
MC       CLR  R0            Set register 0 to 0.
         LI   R1,2          Get the second argument.
         LI   R2,BUF        This is where it goes.
         BLWP @STRREF       Get the string.
         DEC  R1            Return the string in 1st argument.
         BLWP @STRASG       Send the string back now.
         RT                 Now return to the Basic program.
         END
```

To assemble, send the object to "DSK1.MCEAOBJ". The listing can go to your printer or to "DSK1.MCEAL".

The assembly results in one error which can be identified in several ways. First, there is the message "UNDEFINED SYMBOL - 0006". The "0006" refers to line 6 in the program, which has the symbol "BUF". Second, after the program listing there is the heading "THE FOLLOWING SYMBOLS ARE UNDEFINED:" and below that "BUF" is listed. Finally, because the "S" option was used, the symbols are listed by the assembler, and "BUF" is flagged with "U" for "undefined".

From this, we ought to realize that BUF has to be defined. BUF is supposed to be an area of memory: 1) into which the string is copied from the second argument, and 2) from where it is to be copied to the first argument. Memory can be assigned by using the BSS directive. Below is the program with BUF defined.

```
         TITL 'MC - Multiple Concatenate'
         DEF  MC
         REF  STRREF,STRASG
MC       CLR  R0            Set register 0 to 0.
         LI   R1,2          Get the second argument.
         LI   R2,BUF        This is where it goes.
         BLWP @STRREF       Get the string.
         DEC  R1            Return the string in 1st argument.
         BLWP @STRASG       Send the string back now.
         RT                 Now return to the Basic program.
BUF      BSS  255           Define buffer space.
         END
```

This time it assembles with no errors. Now, let's run it to see what happens. (No guarantees!!!)

Running the Basic program results in the message "UNKNOWN ERROR CODE IN 150". A lot of help that is! Unfortunately, that is all too often what happens with assembly language. Remember that one of the strong points of a language like Basic is the

degree of support an interpreted language is able to give to the programmer.

Instead of trying to figure out what that message means, I realized that I forgot to prepare BUF properly before calling STRREF. BUF is to receive a string, and it is necessary to indicate the length of the longest allowable string that can be accepted. This is explained in this book in the chapter on calling from Basic programs, in the TI Editor/Assembler manual, page 287, and in the Mini Memory manual, page 54.

Because the maximum length was not placed in the first byte, the program has to be changed again in two ways. One change is to make BUF one byte longer in order to hold the maximum length in the first byte. A second change is to insert some code to move the maximum length into the byte. Note that it would usually not be good to load the proper value with a DATA or BYTE directive, since the byte is changed to the actual length by STRREF. This means that on subsequent calls to MC, the byte may no longer allow as long a string as we would like. Here is the modified program.

```
        TITL 'MC - Multiple Concatenate'
        DEF  MC
        REF  STRREF,STRASG
MC      SETO @BUF          Allow 255 byte string.
        CLR  R0            Set register 0 to 0.
        LI   R1,2          Get the second argument.
        LI   R2,BUF        This is where it goes.
        BLWP @STRREF       Get the string.
        DEC  R1            Return the string in 1st argument.
        BLWP @STRASG       Send the string back now.
        RT                 Now return to the Basic program.
BUF     BSS  256           Define buffer space.
        END
```

Notice that SETO was used to put 255 into the first byte of BUF. It does this because 255 decimal is FF in hex, which is 1111 1111, or all 1's in binary. And SETO does just that, it sets all the bits to 1's. Now of course it also sets all the bits of the second byte of BUF to 1's as well, but it probably won't matter what is in the second byte, since STRREF is going to copy data on top of the second byte anyway. It would of course also have been possible to use MOVB to move a byte containing 255 into BUF, like this.

```
        MOVB   @MAX,@BUF
        ...
MAX     BYTE   255
```

The program with the above changes assembles with no errors. What happens when it is called? IT WORKS! The string MC$ is set

to "TEST" and is printed. Of course, it is a rather simple program, but nevertheless it is rewarding when it does finally work.

Now it is time to make it more complicated and have it do more than one string. Rather than trying to have it concatenate them, as the next step it would be nice to correctly determine how many arguments there are, and then perhaps just copy the last one. To prove that just the last one is being copied, modify the Basic program to look like this.

```
100 REM  TEST MULTIPLE CONCATENATION
110 CALL INIT
120 CALL LOAD("DSK1.BSCSUP","DSK1.MCEAOBJ")
130 CALL LINK("MC",MC$,"FIRST","SECOND","THIRD")
140 PRINT MC$
150 END
```

Incidentally, if this Basic program calls the current version of MC, MC$ will be set to FIRST. After MC is changed, MC$ should be set to THIRD.

In order to have the assembly language program pick up just the last argument, it is necessary to know how many there are. The description of LINK (both in the TI Editor/Assembler manual and in the Mini Memory Module manual) states that location >8312 contains the number of arguments. This fact can be verified from Basic by inserting these two statements into the Basic program above. The value 4 will be printed for NARGS. (Note that hex 8312 is decimal -31982.)

```
131 CALL PEEK(-31982,NARGS)
132 PRINT NARGS
```

Now to modify MC to use this same fact. This is the changed version of MC.

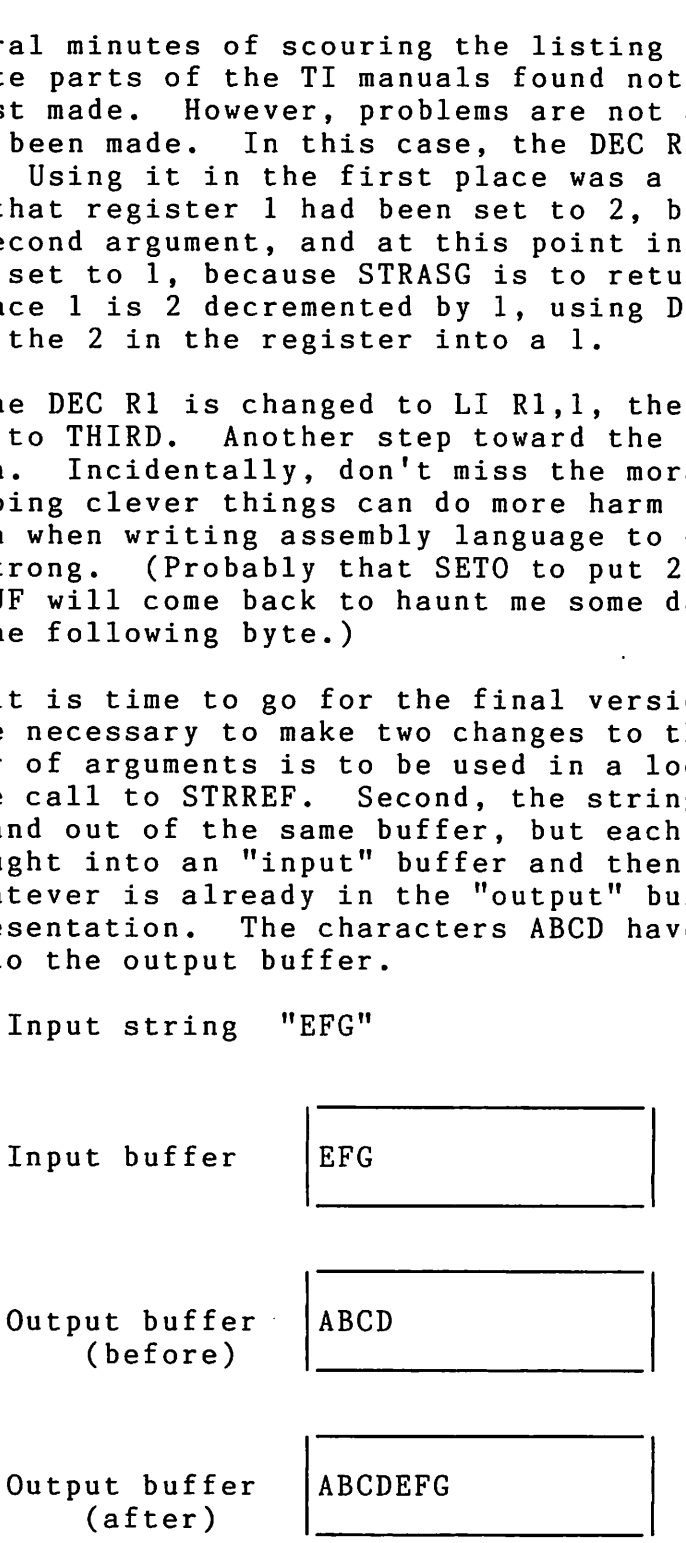
```
        TITL 'MC - Multiple Concatenate'
        DEF  MC
        REF  STRREF,STRASG
MC      SETO @BUF          Allow 255 byte string.
        CLR  R0            Set register 0 to 0.
        CLR  R1            Prepare to move no. args into R1.
        MOVB @>8312,R1     (Value now in left of register)
        SWPB R1            (Value now in right of register)
        LI   R2,BUF        This is where it goes.
        BLWP @STRREF       Get the string.
        DEC  R1            Return the string in 1st argument.
        BLWP @STRASG       Send the string back now.
        RT                 Now return to the Basic program.
BUF     BSS  256           Define buffer space.
        END
```

Well, this version didn't work. When the Basic program called it, the message "BAD ARGUMENT IN 130" appeared.

Several minutes of scouring the listing and rereading the appropriate parts of the TI manuals found nothing wrong with the change just made. However, problems are not always where changes have just been made. In this case, the DEC R1 looks a little mysterious. Using it in the first place was a bad idea. The notion was that register 1 had been set to 2, because STRREF was to get the second argument, and at this point in the program it should be set to 1, because STRASG is to return the first argument. Since 1 is 2 decremented by 1, using DEC seemed a nice way to change the 2 in the register into a 1.

If the DEC R1 is changed to LI R1,1, the program works and A$ is set to THIRD. Another step toward the final version has been taken. Incidentally, don't miss the moral of this little story. Doing clever things can do more harm than good, and the temptation when writing assembly language to do clever things can be very strong. (Probably that SETO to put 255 into the first byte of BUF will come back to haunt me some day because it also changes the following byte.)

Now it is time to go for the final version of the program. It will be necessary to make two changes to the program. First, the number of arguments is to be used in a loop, instead of with the single call to STRREF. Second, the strings cannot be simply moved in and out of the same buffer, but each string must be first brought into an "input" buffer and then stuck onto the back end of whatever is already in the "output" buffer. Here's a visual presentation. The characters ABCD have already been copied into the output buffer.



Below is the modified program. I made two mistakes, which I am not going to show with separate listings. The first I real-

ized before I assembled it: I forgot to change BUF to IN and to define OUT. The second I did not realize until I assembled it and got an error message, letting me know that I did not define the label END before using it in line 12.

```
        TITL 'MC - Multiple Concatenate'
        DEF  MC
        REF  STRREF,STRASG
MC      CLR  R4             Prepare to move no. of args into R4
        MOVB @>8312,R4      (Value now in left of register)
        SWPB R4             (Value now in right of register)
        LI   R3,1           Register 3 will incr. across args.
        LI   R7,OUT         Register 7 is pointer to next
        INC  R7             output byte position for moves.
TOP     INC  R3             Loop across arguments.
        C    R3,R4          Test for last argument.
        JGT  END            Go if just did last argument.
        SETO @IN            Allow 255 byte string on input.
        CLR  R0             Set register 0 to 0.
        MOV  R3,R1          Set register 1 to say which arg.
        LI   R2,IN          This is where it goes.
        BLWP @STRREF        Get the string.
        LI   R6,IN          Register 6 is pointer to next
        INC  R6             input byte position for move.
        CLR  R5             Setup register 5 to be counter
        MOVB @IN,R5         for move of string from IN to OUT.
        JEQ  TOP            If zero length string, skip now.
        SWPB R5             Put length in right of word.
MOV     MOVB *R6+,*R7+      Move a byte from IN to OUT.
        DEC  R5             Count it.
        JGT  MOV
        JMP  TOP            Go back for the next input string.
END     LI   R1,1           Return the string in 1st argument.
        LI   R2,OUT         Compute total length in OUT.
        S    R2,R7          Subtract address of OUT from
        DEC  R7             address of next byte, then
        SWPB R7             subtract 1 more.
        MOVB R7,@OUT        Put result in OUT.
        BLWP @STRASG        Send the string back now.
        RT                  Now return to the Basic program.
IN      BSS  256            Define input buffer space.
OUT     BSS  256            Define output buffer space.
        END
```

This program returns FIRSTSECONDTHIRD. In other words, it works. Since a zero length string is a special case, I changed SECOND to a null string and tried it again. That returned FIRSTTHIRD, which is correct. Aside from the fact that the program lacks tests to make sure that the total length of the concatenated strings is not greater than 255 characters, it is now a useable routine.

A Case History

The big question now is how much speed will it earn for the Basic program which uses it. The figures are interesting. (See the figure below.) I reported the two values in the first line at the beginning of this chapter. When I used the Basic program shown above and called LINK for MC 100 times, it took 19 seconds. That is only a 3 second improvement over the straight Basic version of 22 seconds, a result that is rather disappointing.

I tried to find out why there was not a better increase in speed and modified the Basic program to call MC with only one string.

```
170 CALL LINK("MC",MC$,"A")
```

This took 8 seconds to do 100 times, which should be compared to just 1 second for the equivalent simple Basic version. Obviously there is quite a bit of overhead associated with each CALL LINK - approximately .08 seconds each time.

Times in Seconds for 100 Iterations

	Time To Do Single String	Time To Do All 10 Strings	Time To Do Last String
Using & for Concatenation	1	22	N/A
Link for MC	8	19	17

Subtracting the 8 seconds (for the calls to LINK) from the 19 second total for 100 iterations leaves 11 seconds for the actual concatenation. Because I wanted to try to determine how much of this 11 seconds was due to Basic preparing to call the subroutine and how much was in the subroutine itself, I temporarily modified the subroutine by placing LI R4,2 after SWPB R4 (near the front of the program). This makes the program only return the first string, no matter how many are passed in the CALL LINK.

When I ran the program so it would only return the first string, it took only 17 seconds instead of 19. That means that .02 seconds is required for the assembly language routine on each call with all 10 strings being passed to it but only one being returned. Therefore, of the 11 second difference between the 17 seconds required to pass 10 strings and the 8 seconds required to pass only one string, 2 seconds is used in the routine and the remaining 9 must be used by Basic in preparing the argument list for the CALL LINK. This is approximately .01 second per argument.

A Case History

Perhaps a real increase in speed over Basic cannot be achieved in a case like this unless the number of arguments is kept to a minimum. The easiest way to keep the number of arguments to a minimum yet still refer to a number of strings is to use a string array. This is the Basic program after being modified to use such an array. Notice the call to LINK in statement 210; the S$() is the correct way to pass a string array to a subroutine.

```
100 REM TEST MULTIPLE CONCATENATION
110 DIM S$(11)
120 CALL INIT
130 CALL LOAD("DSK1.BSCSUP","DSK1.MCEAOBJ")
140 FOR I=1 TO 10
150 READ S$(I)
160 NEXT I
170 DATA A,B,C,DEF,GHIJKL,MNOPQRSTUV,W,X,Y,Z
180 S$(11)=CHR$(0)
190 INPUT N
200 FOR I=1 TO N
210 CALL LINK("MC",MC$,S$())
220 NEXT I
230 PRINT MC$
240 END
```

Notice that S$ is dimensioned more than large enough to take all the strings to be copied into it. This is so that a special value can be used to mark the end of the data. That value, a null character, is assigned to S$(11) in statement 180. This is an easy method of flagging the end of the data. (Another way of course would be to pass to the subroutine the value 10, or however many values are actually in the list. However, rather than get into the issue of how to pass a number to the routine, it's easier for now to mark the end with a special string.)

Below is the listing of the assembly language routine after being modified to accept the string array. The listing actually went through two intermediate versions, which are not shown. Statement 13 was originally CI @IN,>0100. That did not work because the first operand for CI must be a register, and I had tried a memory reference. (This same restrictions applies to AI also.) After changing it to C @IN,@NUL and inserting a definition of NUL below in the program, it assembled. The value for NUL is >0100 because the length is 01 and the value is 00.

Trying to run it produced the message BAD ARGUMENT IN 210. Once again, the problem turned out to be someplace other than where a change had been made. Down at END is a sequence that passes the resulting string back to Basic. That code originally did not assign a value to register 0 because it still had the correct value (that is, 0) in the original program left over from the calls to STRREF. However, in the revised program, register 0

gets values other than 0. Therefore, when a nonzero value was in register 0 and STRASG was called, it caused the error. The solution is simply to clear register 0 before calling STRASG. After this change is made, the program works.

```
        TITL 'MC - Multiple Concatenate'
        DEF  MC
        REF  STRREF,STRASG
MC      CLR  R3             Register 3 will incr. across array
        LI   R7,OUT         Register 7 is pointer to next
        INC  R7             output byte position for moves.
TOP     INC  R3             Loop across arguments.
        SETO @IN            Allow 255 byte string on input.
        MOV  R3,R0          Set register 0 to index of string.
        LI   R1,2           Set register 1 to 2 second arg.
        LI   R2,IN          This is where it goes.
        BLWP @STRREF        Get the string.
        C    @IN,@NULL      Quit if string was single null chr
        JEQ  END
        LI   R6,IN          Register 6 is pointer to next
        INC  R6             input byte position for move.
        CLR  R5             Setup register 5 to be counter
        MOVB @IN,R5         for move of string from IN to OUT.
        JEQ  TOP            If zero length string, skip now.
        SWPB R5             Put length in right of word.
MOV     MOVB *R6+,*R7+      Move a byte from IN to OUT.
        DEC  R5             Count it.
        JGT  MOV
        JMP  TOP            Go back for the next input string.
END     CLR  R0             Return the string
        LI   R1,1           in the first argument.
        LI   R2,OUT         Compute total length in OUT.
        S    R2,R7          Subtract address of OUT from
        DEC  R7             address of next byte, then
        SWPB R7             subract 1 more.
        MOVB R7,@OUT        Put result in OUT.
        BLWP @STRASG        Send the string back now.
        RT                  Now return to the Basic program.
NULL    DATA >0100          CHR$(0)
IN      BSS  256            Define input buffer space.
OUT     BSS  256            Define output buffer space.
        END
```

This program works and is twice as fast as the original Basic version. Instead of requiring 22 seconds for 100 iterations, it only takes 11 seconds. This is close to what could be calculated from the above experiments: 8 seconds to CALL LINK, 2 seconds for the 2 arguments, and 2 seconds in the subroutine itself.

To make this work from Extended Basic, remove "DSK1.BSCSUP" from the call to LOAD in the Basic program. In the assembly lan-

guage program, replace the REF statement with the appropriate EQU statements: >2010 for STRASG and >2014 for STRREF.

To make this work with the Dow Editor/Assembler, remove the call LOAD from the Basic program. In the assembly language program, remove the TITL, DEF, REF, BSS, and END statements. Put EQU statements at the end for STRREF (>604C), STRASG (>6048), and IN and OUT. Make an entry in the REF/DEF table for MC. Choose locations in CPU RAM for IN and OUT so that they are within the limits of the 4K RAM but beyond the program itself. They cannot be assigned locations with BSS because of the size limitation on each program segment with the Dow Editor/Assembler. For example, you might load the program at >7118 and put IN at >7200 and OUT at >7300. Finally, shorten all labels to three or fewer characters, change commas to semicolons, and replace RT with B* R11.

Sorting

One of the useful things a computer can do is rearrange data. One method for doing this is to put values into order. This is called **sorting.**

To sort requires making many comparisons of values in a list. If the list is long, there are many many comparisons. Since Basic is slow, sorting in Basic can be painfully slow, but Assembly language really shines when used for this type of task.

The programs in this chapter use a type of sort called a **Shell sort.** Although the programs are rather small, they are difficult to understand unless you work on it for a while. Here is an example of how the algorithm works when sorting 22 random numbers.

```
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
31 41 59 26 53 23 23 54 34 93 28 49 39 28 39 11 39 83 29 84 32 01
```

The first step is to think of the list as 11 pairs, identified with the letter A-K, like this.

```
   1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
A 31                                 49
B    41                                 39
C       59                                 28
D          26                                 39
E             53                                 11
F                23                                 39
G                   23                                 83
H                      54                                 29
I                         34                                 84
J                            93                                 32
K                               28                                 01
```

Now, make sure each pair is in order. This is the result. (The pairs B, C, E, H, J, and K had to be switched.)

```
   1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
A 31                                 49
B    39                                 41
C       28                                 59
D          26                                 39
E             11                                 53
F                23                                 39
G                   23                                 83
H                      29                                 54
I                         34                                 84
J                            32                                 93
K                               01                                 28
```

In the above operation, each pair is separated by 11 items, which is half the number of items to be sorted. In the next step, merge these pairs into five lists of four or five numbers, using an interval of 5 (which is 11 divided by 2). The lists are A-E.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
A	31					23					01					53					93	
B		39					23					49					39					28
C			28					29					41					83				
D				26					34					59					54			
E					11					32					39					84		

Next, sort these five lists. Do the first five pairs first. Start at the left, in row A, and compare items 1 and 6 (31 and 23). They are out of order, so switch them. Then go to row B and compare items 2 and 7 (39 and 23). They are also out of order, so switch them too. There is nothing to do for the pairs in the next three rows (3 and 8 in C, 4 and 9 in D, and 5 and 10 in E), since they are all in order. At this point, it looks like this.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
A	23					31					01					53					93	
B		23					39					49					39					28
C			28					29					41					83				
D				26					34					59					54			
E					11					32					39					84		

Now go back to row A to compare the pair at 6 and 11 (31 and 01); the two values are out of order. After making this switch, move to the left on line A and compare items 1 and 6 (23 and 01). Since these also are out of order, switch them. The first three items on line A are now 01, 23, and 31. Continue like this for all the remaining numbers. This is the rule: whenever a pair is switched, continue to move across the same line to the left, switching again and again until a pair is in order. At that point, go back to where you started switching and continue moving to the right. It should look like this when done.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
A	01					23					31					53					93	
B		23					28					39					39					49
C			28					29					41					83				
D				26					34					54					59			
E					11					32					39					84		

The next step is to repeat the above procedure, stepping across by 2 (which is half of 5). These are the lists A and B before sorting.

```
   1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
A 01     28     11     28     34     31     41     39     39     59     93
B    23     26     23     29     32     39     54     53     83     84     49
```

Compare 01 and 28; they are in order, so do not switch. Compare 23 and 26; they also are in order, so do not switch. Compare 28 and 11; they are out of order, so switch, then compare 01 and 11 - they are in order, so do not switch. Continue in this manner until done. It should look like this.

```
   1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
A 01     11     28     28     31     34     39     39     41     59     93
B    23     23     26     29     32     39     49     53     54     83     84
```

These lists are again merged, and the process repeated with the interval of 1 (which is half of 2). The merged list is this.

```
   1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
01 23 11 23 28 26 28 29 31 32 34 39 39 49 39 53 41 54 59 83 93 84
```

Compare 01 and 23; in order, do not switch. Compare 23 and 11; out of order, switch, then compare 01 and 11. Compare 23 (in position 3, just moved from position 2) to 23 (in position 4); in order, do not switch. Compare 23 and 28, and so forth. This is the final list, now completely sorted.

```
   1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
01 11 23 23 26 28 28 29 31 32 34 39 39 39 41 49 53 54 59 83 84 93
```

That is <u>how</u> the sort works, although <u>why</u> it works is beyond the scope of this book.

Sorting

Here is a Basic program which does this type of sort. Try it with small values of N just to see how it works. The program prompts for N, then generates N random values. It tells you to press ENTER before starting the sort so you can time it. Stop timing when you see DONE. It will then print the sorted values. I timed it with N = 1000, and it took 598 seconds.

```
100 REM  SHELL SORT IN BASIC
110 DIM A(1000)
120 INPUT N
130 FOR I=1 TO N
140 A(I)=RND
150 NEXT I
160 INPUT "PRESS ENTER TO START":T$
170 REM  START SORT
180 INTRVL=N
190 REM  (TOP)
200 INTRVL=INT(INTRVL/2)
210 IF INTRVL=0 THEN 380
220 STOPVAL=N-INTRVL
230 P=1
240 REM  (NXT)
250 LPTR=P
260 REM  (BCK)
270 RPTR=LPTR+INTRVL
280 IF A(LPTR)<=A(RPTR)THEN 340
290 T=A(LPTR)
300 A(LPTR)=A(RPTR)
310 A(RPTR)=T
320 LPTR=LPTR-INTRVL
330 IF LPTR>0 THEN 260
340 REM  (OK)
350 P=P+1
360 IF P>STOPVAL THEN 190
370 GOTO 240
380 REM  (END)
390 REM  SORT COMPLETE
400 INPUT "DONE":T$
410 FOR I=1 TO N
420 PRINT A(I)
430 NEXT I
```

The names in parentheses, such as (BCK) in statement 260, are the same as the labels which appear in the assembly language version below. Also, the logic of the sort is identical, so you can compare this Basic program with the assembly language code. There are two small differences: the variables P and STOPVAL are identified as POS and STOP in the assembly language routine.

Here now is a Basic program to call such a sort, using the Mini Memory Module and the Dow Editor/Assembler. (The version of the sort using Extended Basic will come later because it is slightly more complicated.) This takes 78 seconds to sort 1000 values, which is almost 8 times faster than the Basic version listed above. When you press ENTER after it prints DONE, it will not display the entire list but just display some values from middle of the list as well as the first and last values.

```
100 REM TEST SORT SUBROUTINE
110 DIM A(1000)
120 CALL CLEAR
130 INPUT N
140 FOR I=1 TO N
150 A(I)=RND
160 NEXT I
170 INPUT "PRESS ENTER TO START":T$
180 CALL LINK("SORT",A(),N)
190 INPUT "DONE":T$
200 FOR I=N/2-10 TO N/2+10
210 PRINT A(I)
220 NEXT I
230 PRINT A(1),A(N)
240 GOTO 130
```

This is the Dow Editor/Assembler version of the sort. The numbers in the left margin are mentioned in the explanation in the text that follows.

```
1  000      CLR  R0        GET N; 2ND ARGUMENT
   002      LI   R1;2
   006      BLWP @REF      FAC = N
2  00A      MOV  @FAC;R2   CONVERT N TO INTEGER
3  00E      AI   R2;>C000  SUBTRACT EXPONENT = >4000
4  012      CI   R2;>0100  IF RESULT < 0100;
5  016      JLT  L1        THEN R2 IS ALREADY = N.
6  018      AI   R2;>FF00  R2 STILL TOO BIG. SUBTRACT 0100.
7  01C      MPY  @HUN;R2   MULTIPLY VALUE SO FAR BY 100.
8  020      CLR  R2        NOW GET SECOND BYTE.
9  022      MOVB @FA2;R2
10 026      SWPB R2        ADD PRODUCT FOR FINAL VALUE.
11 028      A    R3;R2
12 02A L1 :MOV  R2;@N
13 02E TOP:SRL  R2;1       INTRVL=INTRVL/2
14 030      JEQ  END
15 032      MOV  @N;R3     STOP=N-INTRVL
   036      S    R2;R3
16 038      LI   R8;1      POS=1
17 03C NXT:MOV  R8;R6      LPTR=POS
18 03E BCK:MOV  R2;R7      RPTR=LPTR+INTRVL
   040      A    R6;R7
19 042      MOV  R6;R0     COMPARE A(LPTR) TO A(RPTR).
```

```
    044      LI    R1;1
    048      BLWP  @REF
20  04C      MOV   @FAC;@TMP
    052      MOV   @FA2;@TM2
    058      MOV   @FA4;@TM4
    05E      MOV   @FA6;@TM6
21  064      MOV   R7;R0
    066      BLWP  @REF
22  06A      LI    R4;TMP
    06E      LI    R5;FAC
    072      LI    R9;4       INITIALIZE TO LOOP
    076 L3  :C     *R4+;*R5+ COMPARE WORDS
    078      JL    OK
    07A      JNE   SWI
    07C      DEC   R9
    07E      JGT   L3
    080      JMP   OK
23  082 SWI:MOV   R6;R0    OUT OF ORDER.  SWITCH
    084      BLWP  @ASG
    088      MOV   @TMP;@FAC
    08E      MOV   @TM2;@FA2
    094      MOV   @TM4;@FA4
    09A      MOV   @TM6;@FA6
    0A0      MOV   R7;R0
    0A2      BLWP  @ASG
24  0A6      S     R2;R6    LPTR=LPTR-INTRVL
    0A8      JGT   BCK      IF LPTR>0; GO BACK
25  0AA OK  :INC   R8       IN ORDER.  MOVE TO RIGHT.
    0AC      C     R8;R3
    0AE      JGT   TOP      IF POS>STOP GO TO TOP.
    0B0      JMP   NXT      GO TO NEXT.
26  0B2 END:B     *R11
    0B4 HUN:DATA  100
    0B6 N   :BSS  2
    0B8 TMP:BSS   2        TMP (8 BYTES)
    0BA TM2:BSS   2
    0BC TM4:BSS   2
    0BE TM6:BSS   2
    0C0 FAC:EQU   >834A
    0C2 FA2:EQU   >834C
    0C4 FA4:EQU   >834E
    0C6 FA6:EQU   >8350
    0C8 REF:EQU   >6044    NUMREF
    0CA ASG:EQU   >6040    NUMASG
```

The registers are used as follows.

```
R0   for NUMREF and NUMASG
R1   for NUMREF and NUMASG
R2   INTRVL: N/2, then half that, and half that, etc.
R3   STOP value for each iteration = N - R2
R4   pointer to A(LPTR) for comparison.
R5   pointer to A(RPTR) for comparison.
R6   LPTR
R7   RPTR
R8   POS
R9   miscellaneous loop counter
```

At 1, call NUMREF to move the value of N into FAC. Then, at through 12, convert it to an integer. Go back to the discussion in the chapter on calling from Basic to see what numbers look like internally. Remember that the only legitimate values of N for this subroutine would be between 0 and 1000 or so (and certainly never a five digit value). These values all require at most 2 or 3 bytes when stored in Basic, a fact which makes it easy to convert them to integers for use in an assembly language program.

Move the first two bytes into register 2. The first byte is the exponent and must be either 64 or 65. At 3, subtract 64 (which is >4000 in hex). The register should then hold >00xx or >01xx. So, at 4, compare to see which it is. If >00xx, the value has been converted, so go to L1 (at 5). Otherwise, it is >01xx, so subtract off the >0100 at 6, then (at 7) multiply the second byte (now left in the right half of the register) by 100. The product is now in register 3. At 8 through 10, move the third byte of FAC (which is the second byte of the value) into register 2 and flip to the right side. At 11, add the product (of the first byte and 100) into register 2.

At 12, move the converted value of N from register 2 into N.

Now, at 13, is the top of the sort loop. It has the label TOP. At this point, the list is essentially split into multiple lists, based on a new interval. In the examples of sorting above in this chapter, the various sublists were shown on different lines for clarity, but of course in memory this is not actually done. The SRL R2;1 computes the new value for the interval. The first value is N/2.

At 14, jump if equal to 0 to END. In other words, if the interval just computed by shifting is 0, the sort is done.

Otherwise, at 15, set register 3 equal to the stopping value for the position as you scan across the list, checking pairs for proper order; STOP = N - INTRVL. For example, in the example in this chapter, on the first iteration the interval is 11, so the

last possible value for the left member of a pair is 22 - 11 = 11. For the second iteration, the last possible value is 22 - 5 = 17. For the next iteration, the last possible value is 22 - 2 = 20. And obviously, on the last iteration, the value is 22 - 1 = 21.

Now, at 16, start the scan at the left by setting register 8 (POS) to 1.

At 17 the label is NXT. Here prepare to compare the items at LPTR and RPTR. LPTR is the pointer to the left value of the pair, and RPTR is the value on the right. LPTR, register 6, is initially set to POS.

At 18, compute RPTR = LPTR + INTRVL. That is, register 7 is the sum of register 6 and 2. To actually compare the values at LPTR and RPTR, at 19 move LPTR into register 0, set register 1, and call NUMREF. Because LPTR is in register 0, this moves A(LPTR) into FAC.

At 20, move the eight byte value from FAC into TMP. This is done with four MOV instructions. Because the Dow Editor/Assembler does not allow addresses such as FAC+2, it is necessary to define symbols such as FA2 with EQU directives. The same is done for TMP, TM2, and so forth. Four MOV instructions will move the eight bytes.

At 21, move RPTR (register 7) into register 0 and call NUMREF again to get A(RPTR) in FAC.

Now, at 22, both values are available. Make register 4 point to TMP and register 5 point to FAC. Initialize register 9 to be 4, then loop to compare four words. The top of the loop is at the label L3 and the bottom is at the statement JGT L3. If the first value is less than the second, the numbers are in order and the JL will cause a transfer to OK. (Because this is JL rather than JLT, the sort is a **logical** rather than a **numeric** sort. That is negative values will not sort before positive values.) If the values are not equal, the first must be greater than the second, so transfer to SWI to switch them. Otherwise, decrement register 9 and jump back to L3 if register 9 is still greater than 0. Finally, if the two numbers were the same for all four words, go to OK.

At 23, switch the two values. This is done by calling NUMASG to store them back into A. Reverse the order of register 6 and 7 to switch the values.

At 24, because the values were out of order and had to be switched, it is necessary to move to the left in the list. Therefore, subtract INTRVL (register 2) from LPTR (register 6). If this results in a value still within the list (i.e., greater

than 0), go to BCK to compare again.

At 25, move POS one step to the right. If POS is now greater than STOP, the iteration has been completed, so go to TOP. Otherwise, go to NXT go compare pairs some more.

At 26, the program has finished, so return to Basic. Next comes the definition of HUN, which is the value 100; this is used for multiplication during conversion of N. Then come TMP, TM2, TM4, and TM6 (eight bytes to hold a value during comparison). After that are the EQU statements for FAC and so forth.

Notice that the conversion logic does not test for N = 0, and the program will do bad things in this case. Also, you may be able to make this routine faster by not using NUMASG to store both members of the pair after switching, since one is to be used immediately in the next comparison.

From the sort programs shown above in both Basic and assembly language, you can see the sort algorithm itself. Here is another version which uses exactly the same algorithm but which relies on the use of the 32K memory expansion. It is shown for use with Extended Basic, since it depends on the numerical array being stored in the 32K memory.

The reason that the memory expansion is required is that the program can directly access the array in that memory, without having to send data back and forth between VDP RAM and CPU RAM. Even though this transfer can be done conveniently with the utility routines NUMREF and NUMASG, it is rather slow. Because of this change in the logic, the program can sort 1000 values in just 5 seconds. This is 15 times faster than the previous version, and 120 times faster than the Basic version. (It is capable of sorting 3000 values in just 20 seconds.

Here is the Basic program to call the modified assembly language sort routine.

```
100 REM TEST XBSORT SUBROUTINE
110 DIM A(3000)
120 CALL CLEAR
121 CALL INIT :: CALL LOAD("DSK1.XBSORTOBJ")
130 INPUT N
140 FOR I=1 TO N
150 A(I)=RND
160 NEXT I
170 INPUT "PRESS ENTER TO START":T$
180 CALL LINK("XBSORT",A(),N)
190 INPUT "DONE":T$
200 FOR I=N/2-10 TO N/2+10
210 PRINT A(I)
220 NEXT I
```

```
          230 PRINT A(I),A(N)
          240 GOTO 130
```

By comparison to the Basic program which calls the Mini Memory sort routine, note the addition of statement 121 and the change of SORT to XBSORT in statement 180.

Here is the sort routine modified to use the 32K memory expansion. The numbers in the left margin are used in the text below.

```
          TITL 'XBSORT'
   *
   * CALL LINK("XBSORT",A(),N)
   *
   * REGISTER USAGE:
   * R0 TEMP FOR SUBSCRIPTS ETC
   * R2 INTRVL  N/2 ETC
   * R3 STOP    N-INTRVL
   * R4 POINTER TO A(LPTR)
   * R5 POINTER TO A(RPTR)
   * R6 LPTR
   * R7 RPTR
   * R8 POS
   * R9 MISCELLANEOUS LOOP COUNTER
   * R10 BASE ADDRESS FOR A()
   *
   FAC    EQU  >834A
   NUMREF EQU  >200C
   VMBR   EQU  >202C
   *
          DEF  XBSORT
1  XBSORT LI   R0,16        GET R1=BASE ADDRESS FOR A()
          A    @>8310,R0    C(8310)+16 POINTS TO STACK ENTRY
          LI   R1,STACK                          IN VDP MEM.
          LI   R2,8
2         BLWP @VMBR
3         MOV  @STACK+4,R0  BYTES 4&5 POINT TO ARRAY IN VDP.
          LI   R2,4         VDP HAS 2 BYTES ARRAY SIZE,
          LI   R1,DATA      FOLLOWED BY ADDR OF DATA IN EXP MEM
          BLWP @VMBR
4         MOV  @DATA+2,R10  R10 NOW POINTS TO A(0) IN EXP MEM.
5         CLR  R0           GET N, 2ND ARGUMENT
          LI   R1,2
          BLWP @NUMREF      FAC=N.
          MOV  @FAC,R2      CONVERT N TO INTEGER
          AI   R2,>C000       SUBTRACT EXPONENT = >4000.
          CI   R2,>0100       IF RESULT < 0100,
          JLT  L1             THEN R2 IS ALREADY = N.
          AI   R2,>FF00       R2 STILL TOO BIG. SUBTRACT 0100.
          MPY  @V100,R2       MULTIPLY VALUE SO FAR BY 100.
          CLR  R2             NOW GET SECOND BYTE.
```

```
            MOVB @FAC+2,R2
            SWPB R2
            A    R3,R2          ADD PRODUCT FOR FINAL VALUE.
   L1       MOV  R2,@N
   TOP      SRL  R2,1           INTRVL=INTRVL/2
            JEQ  DONE
            MOV  @N,R3          STOP=N-INTRVL
            S    R2,R3
            LI   R8,1           POS=1
   NEXT     MOV  R8,R6          LPTR=POS
   GOBACK   MOV  R2,R7          RPTR=LPTR+INTRVL
            A    R6,R7
6           MOV  R6,R4          COMPARE A(LPTR) TO A(RPTR).
            SLA  R4,3
            A    R10,R4
7           MOV  R7,R5
            SLA  R5,3
            A    R10,R5
8           LI   R9,8           INITIALIZE TO LOOP.
   L3       CB   *R4+,*R5+      COMPARE BYTES
9           JL   OKAY
            JNE  SWITCH
            DEC  R9
            JGT  L3
            JMP  OKAY
10 SWITCH   MOV  R6,R4          A(RPTR) > A(LPTR)
            SLA  R4,3           SWITCH THE VALUES.
            A    R10,R4
11          MOV  R7,R5
            SLA  R5,3
            A    R10,R5
12          LI   R9,8           INITIALIZE TO LOOP.
   L4       MOVB *R4,R0         SWITCH BYTES
            MOVB *R5,*R4+
            MOVB R0,*R5+
            DEC  R9
            JGT  L4
13          S    R2,R6          LPTR=LPTR-INTRVL
            JGT  GOBACK         IF LPTR>0 GO TO GOBACK
   OKAY     INC  R8             A(LPTR) < A(RPTR). SET POS=POS+1.
            C    R8,R3
            JGT  TOP            IF POS>STOP GO TO TOP
            JMP  NEXT           GO TO NEXT
   DONE     B    *R11
   V100     DATA 100
   N        BSS  2
   STACK    BSS  8              8 BYTE STACK VALUE FOR A()
   DATA     BSS  4              A() DIM & PTR TO EXP MEMORY.
            END
```

This assembly language program differs from the Dow Editor/Assembler version above in the standard ways discussed already

in this book. For instance, there are a number of comment lines at the beginning of this version. Also, the EQU value NUMREF is different.

However, the significant difference begins at 1. Extended Basic stores numerical data, such as the array A, in the 32K memory expansion. The first thing this program does is find out where it is stored in this memory. Page 278 of the TI Editor/Assembler manual, while describing LINK, says that location >8310 in CPU RAM contains the value stack pointer. (See also Appendix A of this book.) Unfortunately, that statement is rather too cryptic. However, I determined by experimentation and partially disassembling the ROM for NUMREF that in fact the contents of that location plus 16 does indeed point to an 8-byte pointer for the first argument to LINK.

Therefore, at 1, the contents of locations >8310 and >8311 are added to the value 16 in register 0. The resulting sum is used in a call to VMBR, at 2. **VMBR** is a utility routine which passes data from VDP memory to CPU memory; it is described on page 249 of the TI Editor/Assembler manual and on page 36 of the Mini Memory manual. To use it, register 0 must contain the address in VDP memory from which data is to be read. Register 1 contains the address where it is to be sent (STACK in this case). And register 2 contains the number of bytes to transfer (8 in this case).

After calling VMBR, STACK holds the 8-byte stack value for A. Stack values are described in the TI Editor/Assembler manual on page 280 and on pages 27 and 28 of the Mini Memory manual. For a numeric array, bytes 4 and 5 are the address in VDP memory where the array is stored. The manual neglects to point out that if Extended Basic is in use, the array is not actually in VDP memory. However, VDP memory at that location does contain size information followed by a pointer into CPU memory. So, at 3, move the pointer (to VDP memory) into register 0 and then call VMBR again to read 4 bytes into DATA. At 4, the third and fourth bytes thus obtained are the address of A in CPU memory; this address is moved into register 10 for later use.

At 5, the program obtains N and converts it to an integer, just as in the other version.

At 6, the programs differ. The other version had to pull A(LPTR) and A(RPTR) from VDP memory into FAC, using NUMREF. This version compares the two values while they are actually in the array, and exchanges them in place. To do this, it needs to add the value of LPTR to the address in register 10 to come up with the actual address in CPU memory of A(LPTR). At 6, register 6 (LPTR) is moved into register 4, then shifted left 3 bits. This shift has the effect of multiplying it by 8. This needs to be done because there are 8 bytes per numeric value in the array.

Next, register 10 is added to register 4, thus completing the computation of the actual address of A(LPTR). A similar computation is performed for A(RPTR)

At 7, the same type of computation is carried out for RPTR, using registers 7 and 5.

Now, at 8, the two 8-byte quantities need to be compared. Register 9 is loaded with the value 8 to compare the 8 bytes. Notice that the other assembly language program was able to compare 4 words, while this one needs to compare 8 bytes. The other one could compare words because the values were moved into CPU memory at addresses known to be even. However, Extended Basic does not necessarily store numeric data at even addresses, so byte instructions must be used.

Starting at 9, the comparison logic is the same as the other version.

However, at 10 there is again a difference. As before, register 4 is computed to point to A(LPTR), and at 11, register 5 is made to point to A(RPTR). These two 8-byte values in A now need to be exchanged. At 12, there is a loop which exchanges them byte by byte, using register 0 as temporary storage. The MOVB *R4,R0 moves one byte from A(LPTR) to register 0; the MOVB *R5,*R4+ moves the corresponding byte from A(RPTR) into A(LPTR) and increments register 4 to point to the next byte of A(LPTR); and MOVB R0,*R5+ moves the original byte from temporary storage in register 0 into A(RPTR) while simultaneously incrementing the memory pointer for A(RPTR).

The rest of the logic is the same, starting at 13.

Preparing To Sort Names

In the last chapter is a powerful sort routine. It sorts numbers because that is more efficient that sorting strings, but often you want to sort strings instead. For instance, you might want to write a program of your own to create an alphabetical listing of names. This chapter describes an assembly language routine that will pack up to nine letters into one Basic number. You can use it to alphabetize on the first nine characters of the name. In addition to the nine letters, it also packs a reference number in the Basic number so that you can retrieve the original data after sorting.

From the chapter on calling from Basic programs, you should realize that you can in fact store any set of eight bytes you want in a number in Basic. It is possible to compress three letters into two bytes. If the first six bytes are used for letters, that enables the program to pack nine letters, with two bytes left over. The two remaining bytes can be used to hold a pointer to the original data.

This is how to use such a routine for alphabetizing. Suppose you have a RELATIVE file with 1,000 names to be sorted. Write a program to read through the file. As each record is read, the name is pulled out of the record with SEG$ and is placed into NAME$. The routine PACKNM is then called to compress both the name and the record number, REC, into PACKED. Then, PACKED is stored in an array at a position corresponding to the record number. After all records have been read and the names packed, a sort routine cycles down the sorted list, using the compressed record number to retrieve the name from the file for printing.

Here is a simple Extended Basic program which demonstrates the essential steps of this kind of sorting. Since it is just a simple program to demonstrate the technique, it does not read the data from a disk file but accepts it from the keyboard, and it keeps the data in a string array to be printed after sorting.

```
100 REM GENERAL PURPOSE SORT PROGRAM
110 DIM CVEC(100),NAMES$(100)
120 CALL CLEAR
130 CALL INIT
140 CALL LOAD("DSK1.PACKNMOBJ","DSK1.XBSORTOBJ")
150 PRINT "ENTER NAMES."
160 PRINT "(ENTER 'END' WHEN DONE)"
170 LINPUT NAME$
180 IF NAME$="END" THEN 240
190 N=N+1
200 NAMES$(N)=NAME$
210 CALL LINK("PACKNM",NAME$,N,PACKED)
220 CVEC(N)=PACKED
```

```
230 GOTO 170
240 REM ALL DATA HAS BEEN READ. NOW SORT.
250 CALL LINK("XBSORT",CVEC(),N)
260 FOR I=1 TO N
270 PACKED=CVEC(I)
280 CALL LINK("UNPKNM",PACKED,J)
290 PRINT NAMES$(J)
300 NEXT I
310 END
```

Statement 210 passes the name in NAME$ and the 'record number' in N to the routine, and the compressed name and number are returned in PACKED. Statement 250 sorts CVEC, which has the N packed values. Statement 280 gets the record number out of PACKED so it can be used to print the strings in order.

The listing of the assembly language routines starts on the next page. They make up a fairly large program, which should not be written all at once. The sequence I went through to write them was something like this.

First, I simply passed the string and the record number to PACKNM. I converted the record number to an integer and returned it in the first two bytes of PACKED. I checked this carefully before continuing.

Next, I enhanced the routine to move the record number to the last two bytes of PACKED. I also put in logic to map "a" to "A", "b" to "B", and so forth. Characters other than a-z and A-Z were ignored, although a comma was recognized to mean the end of the name. Characters still accepted were moved into TEMP (a holding area within the program). Finally, three words from TEMP were copied into the first three words (six bytes) of PACKED.

After that worked, I added the code to pack three characters into one word. This packing is done by mapping "A" to 1, "B" to 2, and so forth. Since "Z" maps to 26, legitimate values range from 1 to 26. They are packed by multiplying them by powers of 27 and then adding. (The actual algorithm alternates between multplying and adding.) This transforms three letters into a one-word integer value. For example, if the letters are "AAA", the value is 1 * 27 * 27 + 1 * 27 + 1 = 757. If the letters are "ZZZ", the number is 26 * 27 * 27 + 26 * 27 + 26 = 19,862. Since the largest possible value for a one-word integer is 32,767, this technique will not cause any overflow during multiplication.

Finally, I changed it again to map the comma to 0 and to continue with the name past the comma. This causes the program to sort on as much of both the last name and first name as will fit within the nine character limit. Instead of stopping when it hits a comma, it now stops either at the end of the string NAME$ or after packing nine characters.

Preparing To Sort Names

The numbers along the left margin of the listing are mentioned in the text which follows the listing.

```
          SUBROUTINE PACKNM
   *
   *      CALL LINK("PACKNM",NAME$,N,PACKED)
   *      CALL LINK("UNPKNM",PACKED,N)
   *      CALL LINK("UNPKST",PACKED$,N)
   *
   * PACKNM takes a string with up to 36 characters in NAME$
   *        and a number from 0 to 9999 in N and packs them
   *        into PACKED.  Up to 9 characters of the string are
   *        packed into the first 6 bytes of PACKED, followed
   *        by the value N in the last two bytes.  Only the
   *        characters A-Z and a-z and comma are packed, and
   *        A and a are treated alike, B and b, etc.  The comma
   *        sorts before A.
   *
   * UNPKNM returns to Basic the value N that was packed into
   *        the last two bytes of PACKED.
   *
   * UNPKST is the same as UNPKNM except that the input
   *        is a string.
   *
   ************************************************************
   *
   FAC    EQU  >834A
   STRREF EQU  >2014
   NUMREF EQU  >200C
   NUMASG EQU  >2008
   COMMA  EQU  >2C00          =44=","
          DEF  PACKNM,UNPKNM,UNPKST
1  PACKNM MOVB @DB36,@NAME   Get NAME$ in NAME.
          CLR  R0
          LI   R1,1
          LI   R2,NAME
          BLWP @STRREF
2         LI   R1,2          Get N in FAC.
          BLWP @NUMREF
3         MOV  @FAC,R0       Convert floating-point to integer.
4         JEQ  NOSCAL        Done immediately if = 0.
5         AI   R0,>C000      Subtract exponent >4000.
6         CI   R0,>0100      Test for power of 100.
          JLT  NOSCAL        Jump if power of 100 is 0.
7         AI   R0,>FF00      Subtract >0100.
8         MPY  @V100,R0      Scale by 100.
9         CLR  R0            Now get second byte.
          MOVB @FAC+2,R0
          SWPB R0
10        A    R1,R0         Add product for final value.
11 NOSCAL MOV  R0,@FAC+6     Put integer value in FAC+6.
```

Preparing To Sort Names

```
12          LI    R0,NAME       R0 points to NAME
            CLR   R3            Set R3 to length of name.
            MOVB  *R0,R3
            INC   R0            R0 points to first char of name.
            SWPB  R3
            A     R0,R3         R3 points just past end of name.
13          LI    R1,16         Clear TEMP.
   LP       CLR   @TEMP(R1)
            DECT  R1
            JOC   LP
            CLR   R1            R1 = @ words in TEMP so far.
14 NXTBYT   C     R0,R3         Quit if no more name.
            JEQ   NOMORE
15          CLR   R2            Get next char from NAME.
16          MOVB  *R0+,R2
17          CI    R2,COMMA      Change comma into 0.
            JEQ   ANOTHR
18          SWPB  R2
            AI    R2,-64        Map A to 1, B to 2, etc.
            CI    R2,27         Test for too large.
            JLT   SKIP
            AI    R2,-32        Scale a to A, b to B, etc.
19 SKIP     MOV   R2,R2
            JLT   NXTBYT        Too small - ignore.
            JEQ   NXTBYT        Too small - ignore.
20          CI    R2,26
            JGT   NXTBYT        Too large - ignore.
            MOV   R2,@TEMP(R1)
21 ANOTHR   INCT  R1            Count another value in TEMP.
            CI    R1,18
            JLT   NXTBYT        Keep on if not yet 9.
22 NOMORE   CLR   R3            R3 is index into FAC=0,2,4.
23 LP1      MOV   R3,R2         R2 is index into TEMP=0,6,12.
            A     R3,R2
            A     R3,R2
24          MOV   @TEMP(R2),R0  Combine each 3 words in TEMP
            MPY   @V27,R0       into each 1 word of FAC.
            MOV   R1,R0
25          A     @TEMP+2(R2),R0
            MPY   @V27,R0
            A     @TEMP+4(R2),R1
26          MOV   R1,@FAC(R3)
27          INCT  R3
            CI    R3,6          Test for end of loop.
            JLT   LP1
28          CLR   R0            All done. Return PACKED.
            LI    R1,3
            BLWP  @NUMASG
            RT
   *
29 UNPKST   CLR   R0            Get PACKED$.
            LI    R1,1
```

```
            LI   R2,TEMP
            MOVB @DB8,@TEMP
            BLWP @STRREF
            MOVB @TEMP+7,@TEMP
            MOVB @TEMP+8,@TEMP+1
            MOV  @TEMP,R1
            JMP  COMMON
30 UNPKNM   CLR  R0             Get PACKED.
            LI   R1,1
            BLWP @NUMREF
            MOV  @FAC+6,R1      R1=integer at end of FAC.
31 COMMON   CLR  R0
            LI   R2,FAC         Set FAC to 0's.
            MOV  R0,*R2+
            MOV  R0,*R2+
            MOV  R0,*R2+
            MOV  R0,*R2
32          DIV  @V100,R0       Divide integer by 100.
            MOV  R0,R0          See if quotient is 0.
            JNE  LARGE          Jump if >99.
33          AI   R1,>4000       Value is less than 100.
            MOV  R1,@FAC
            JMP  RETN
34 LARGE    AI   R0,>4100       Value is >99.
            MOV  R0,@FAC
            SWPB R1
            MOV  R1,@FAC+2
35 RETN     CLR  R0             Return value in STRING$
            LI   R1,2
            BLWP @NUMASG
            RT
36 NAME     BSS  38             Buffer for NAME.
37 TEMP     BSS  18             Temp workspace. 9 chars (words).
   DB2      BYTE 2              String length returned.
   DB8      BYTE 8
   DB36     BYTE 36             Max string length for NAME.
   V100     DATA 100            Value 100 for scaling integers.
38 V27      DATA 27             Value 27 for packing into FAC.
            END
```

At 1, the string is obtained from the first argument in the call to LINK. In this version of the program, up to 36 characters are acceptable. This allows for a long name. Also, if the name is in the first part of the record, it is not necessary to use SEG$ to separate it from the rest of the record, since the routine will stop after packing nine characters even if the rest of the record contains data which could be mistaken for part of the name. By not having to call SEG$, the program will run faster. At 2, the record number is obtained from the second argument. The routines for doing this were described in the chapter on calling from Basic.

At 3, the program moves the first word of the number into register 0 to begin converting it to an integer. There is no problem with a word move instruction here (rather than two byte move instructions) because FAC and registers are both known to have even addresses.

Since the value 0 is represented in Basic by 0's in the first two bytes of FAC, at 4 a JEQ following the MOV is all that is needed to test for 0; if it is 0, go to NOSCAL ("no scaling needed") to use 0 as the converted value. Register 0 already contains 0; at NOSCAL, it is moved from register 0 into the last two bytes of FAC.

If the value is not zero, there is a bias of 64 (hex >40) in the exponent, and this has to be subtracted. Therefore, at 5, >C000 is added to register 0. (The value >C000 is the two's complement of >4000.) Given the range of numbers that can be converted (0 through 9999), the exponent must be 64 or 65. Therefore, after subtracting, it must be either 0 or 1. That is, the first word of FAC must now be either >00xx or >01xx. At 6, go to NOSCAL if the left byte is less than 1 (that is, if it is 0). In this case, because the value is between 1 and 99, having subtracted off the exponent, the first two bytes contain >00xx, where xx is the number itself, so it can now be stored in FAC.

At 7, the number was larger than 99, so the values in two bytes have to be added together. Remove the exponent from the word (it is the left byte and is equal to 1) by subtracting >0100. Then multiply by 100 at 8, which leaves the product in registers 0 and 1. At 9, clear register 0, move the third byte into the left half of the register with MOVB, then switch it to the right half with SWPB. At 10, add the product of the first byte and 100 to the third byte now in register 0, thus completing the conversion.

At 11, move the converted integer from register 0 into the last two bytes of FAC, in preparation for being returned to the Basic program.

At 12, compute the address of the first byte past the end of the input string. Make register 0 point to the first byte, then move the byte (since it contains the length of the string) into the right half of register 3, add 1 to the address in register 0, and then add the address to the length in register 3.

Now, at 13, loop to clear all 18 bytes of TEMP. TEMP has nine words, to hold nine characters temporarily (one character per word to make arithmetic easy). TEMP is defined at 37. Also, initialize register 1 to keep track of the number of characters that have been moved into TEMP.

At 14 is the top of the main loop. It is identified by the

label NXTBYT. Remember that register 3 was computed to be the address of the byte just past the end of the string. Register 0 was left holding the address of the first byte in the string. At 16, register 0 is incremented each time through the loop. Thus, the comparison at 14 is executed each time through the loop. It will be equal when register 0 has been incremented enough to equal register 3. (If the input string is empty, the test will succeed on the first attempt.)

At 15 and 16, move the next character into the left half of register 2, making sure the right half is 0's.

At 17, test the character to see if it is a comma. Note the use of the symbol COMMA, which is not an address but the value >4400, which is the character ",". Jump to ANOTHR if it is a comma. (Because TEMP was filled with 0's initially, this means that each comma is represented by 0 in temp.)

At 18, move the character into the right half of the word, then subtract 64 from it to map "A" into 1, "B" into 2, and so forth. Compare the result to 27; if less than 27, it is okay. Otherwise, subtract another 32 in case the character was a lower case letter.

At 19, move register 2 to itself - this is a means of comparing it to 0. If the register is less than 0 or equal to 0, ignore it by jumping to NXTBYT.

At 20, see if the value is not greater than 26. Jump to NXTBYT if it is greater than 26. If not, move the word (now a value between 1 and 26) into TEMP (indexed by register 1).

The bottom of the loop is at 21. Increment register 1 to count another value moved into TEMP (or a comma). Check to see if register 1 is now 18; if not yet 18, continue to process the input string by going back to NXTBYT.

The compressing of the nine characters into three words begins at 22. Register 3 will loop across the three words of FAC, which get the results.

The top of the compression loop is at 23. Move register 3 into register 2, then add it to register 2 twice more. This has the effect of setting register 2 to three times the value in register 3. (It is actually easier to add several times than it would be to multiply.)

At 24, move a word value from TEMP (indexed by register 2) into register 0. Multiply by 27; V27 is the value 27, defined at 38. Move the product from register 1 back to register 0. At 25, add the next value from TEMP into register 0. Multiply the sum by 27, and add the third value from TEMP into register 1 as well.

At 26, move the combined value into FAC, indexed by register 3.

The bottom of the compression loop is at 27. Increment register 3 to move along FAC, and test to see if it is six yet. If still less than six, go back to LP1 to continue compressing.

At 28, call NUMASG to return the value in FAC to the third argument in the call to LINK. Then return.

The UNPKST entry point is at 29. It is used if the packed data is passed as a string, rather than as a number. It calls STRREF to get an eight character string. Then it moves the two bytes from the string into register 1.

The UNPKNM entry point is at 30. It uses NUMREF to get the number, then moves the integer value at the end of it into register 1.

Whether UNPKST or UNPKNM was called, at 31 the integer has been moved into register 1. It needs to be converted into the eight character form Basic uses. First, set all eight bytes of FAC to 0's. This is done by clearing register 0, setting register 2 to point to FAC, and then executing four MOV's of register 0 (which contains 0) indirect of register 2. Each move increments register 2, thus moving along FAC automatically.

At 32, register 0 and 1 (combined) are divided by 100. The quotient is tested for 0; if not 0, the value is larger than 100, so go to LARGE.

At 33, the value is less than 100, so add in the exponent of 64 (hex >4000), put the result in FAC, and go to RETN to return.

At 34, because the value was larger than 100, add the exponent of 65 (hex >4100) to the quotient and put the sum into the first two bytes of FAC. Then move the remainder from the right half of register 1 into the third byte of FAC.

Finally, at 35, FAC is passed back to Basic as the second argument.

At 36, various symbols are defined and memory space is allocated. NAME, for instance, is given 38 bytes. It needs 36 bytes to hold a string that long; it needs an extra byte to hold the length byte at the front; and the 38th byte is to make it even length.

The Extended Basic program can be converted for use with the TI Editor/Assembler and Basic by including DSK1.BSCSUP in statement 140, changing XBSORT to SORT in 140 and 250, and changing LINPUT to INPUT in 170. For use with the Dow Editor/Assembler, also delete statements 130 and 140.

Interrupts, Screen, and Keyboard

The use of **interrupts** greatly enhances the utility of computers. Without interrupts, a CPU (central processing unit) could in practice only handle one task. To handle more than one task, it would be necessary to write into each program frequent points at which the computer would check to see if something else needed to be done. To make this point clearer, try the following Basic program. (If you do not have Extended Basic, omit the CALL SPRITE; the CALL SOUND alone will provide the demonstration.)

```
100 REM DEMONSTRATE INTERRUPTS
110 CALL CLEAR
120 CALL SPRITE(#1,97,2,20,20,10,10):: CALL
SOUND(1000,500,0)
```

The program will clear the screen, generate a tone, then put a lower case "a" in the upper left corner and move it towards the lower right corner. The "a" is of course a sprite.

Now change the program as shown below to call the small assembly language routine, also shown below.

```
100 REM DEMONSTRATE INTERRUPTS
110 CALL INIT :: CALL LOAD("DSK1.HANGUPOBJ")
120 CALL CLEAR
130 CALL SPRITE(#1,97,2,20,20,10,10):: CALL
SOUND(1000,500,0)
140 CALL LINK("HANGUP")

      DEF    HANGUP
HANGUP JMP   HANGUP
      END
```

Calling this version of the program demonstrates dramatically the importance of interrupts. When you run it, the screen will clear, the tone will start, and the "a" will appear, just as before. However, the sound will never end and the "a" will not move. Furthermore, the computer will be locked up. The only way to regain control is to turn it off. Pressing CLEAR and even pressing QUIT has no effect.

There are two reasons why the little assembly language routine can tie up the computer. The first is that the Basic interpreter turns off interrupts before calling the assembly language routine. The second is that the assembly language routine does not return control to Basic. Taken together, these facts mean that the computer will no longer process interrupts. Without interrupts it cannot move the sprite, stop the sound, or check the keyboard to see if you are pressing CLEAR or QUIT.

You can further prove the importance of interrupts by substituting this assembly language program.

```
        DEF    HANGUP
HANGUP  LIMI   2
        LIMI   0
        JMP    HANGUP
        END
```

In this routine, the **LIMI 2** instruction **enables interrupts**, and the **LIMI 0** immediately **disables interrupts.** (Programmers also talk about "unmasking" and "masking" interrupts.) If an interrupt is disabled or masked, it cannot be "serviced" by the CPU. However, as soon as it is unmasked, it is serviced. In other words, an interrupt will wait indefinitely to be serviced.

A clock in the computer causes the interrupts 60 times per second. Unless the interrupts are masked, the CPU does certain bookkeeping tasks each time. These tasks include tending to sprites and sound generation and checking to see if you are pressing QUIT.

To understand the practical significance of interrupts, even on a home computer which only has to pay attention to one person at a time, imagine writing a complicated and fairly time consuming program so that it would check all the time to see if the user is holding down a key to stop the program. How much simpler it is to have a clock get the computer's attention on a regular basis, without our having to think about it.

This is what happens when there is an interrupt that is not masked (or when an interrupt that has already occured is ultimately unmasked). Control jumps out of your program to a special location, determined by the designers of the computer. Starting at that location is a routine which determines what caused the interrupt and responds to it properly. This routine is called an **interrupt handler.** For instance, as stated above, if the interrupt was the clock, the routine checks the keyboard to see if you are holding down the QUIT key. On a larger computer, there are many things that can cause interrupts, including printers and terminals.

You need to know about interrupts only if you need to have them enabled for sprite movement, for sound, or to allow the user to break out of the program. If you do enable them, you have to be careful how you do it. The best technique is to have the LIMI 0 come immediately after the LIMI 2 at a place in the program that is executed fairly often. That is, unmask them frequently, but do not leave them unmasked all the time.

The reason you do not want to leave them unmasked all the time is that in that case you have no idea when an interrupt will occur. Due to the manner in which VDP and GROM are accessed, a program should not be interrupted during the transfer of data between either of these memories and CPU memory. If the interrupt processor also uses these other memories, the hardware will "forget" your location in the memory and transfer data to or from the wrong place. This can have disastrous consequences for your program.

There is no harm in executing the LIMI 2 and LIMI 0 instructions if there is no interrupt pending. The TI Editor/Assembler manual suggests that a good time to allow interrupts is at the same point in your program where you are checking the keyboard to see if the user is pressing a key. (This would be done fairly frequently if you were writing some kind of game or simulation program.)

Screen manipulation is quite different in assembly language than in Basic. For example, there is no simple way to clear the screen, as Basic does it with just one statement - CALL CLEAR. There is no PRINT statement which causes the screen to scroll automatically. Furthermore, in a Basic program you put data on the screen using subroutines such as HCHAR, VCHAR, and (in Extended Basic) DISPLAY; in all of these, the location on the screen is defined by a row and column coordinate. However, in assembly language you must compute the coordinate yourself from the row and column. (Incidentally, this step is necessary even in Basic on some other computers.)

In the normal screen mode used by Basic, called **graphics** mode, there are 32 columns and 24 rows. This means there are 768 positions. Each of the positions is defined by a byte in VDP memory. For instance, the upper left corner of the screen is defined by location 0 in VDP memory. Locations 0 through 31 define the top row of the screen. Suppose you want to put a character at the 5th position of the 20th row. To compute that location in VDP memory, you have to take into account the 19 rows that were skipped. That is 19 times 32 positions, or 608. There are also four positions to be skipped in the 20th row to get to the 5th position. This makes the exact count 612. Thus, row 20 column 5 is position 612.

In order to display a character at a given position, you need to move a byte value into that location in VDP memory. The value of that byte should be the equivalent ASCII value for the character you want to display. Suppose for instance that you want to display an "X", which is represented by the value 88. Move 88 into location 612 of VDP memory to put an X in row 20 column 5.

If your routine is called from a Basic program, the value moved into VDP memory must have 96 added to it so the proper character is displayed. For instance, put 184 into VDP memory to display an X. (The BTXT directive in the Dow Editor/Assembler does this for your automatically.)

Reading data from the keyboard is also very different in assembly language than the standard INPUT statement in Basic. All assembly language has is **KSCAN**, which is essentially identical to CALL KEY. Notice that KEY in Basic and KSCAN in assembly language can be called when a key is not being pressed, so they do not always return any data. Also, even when data is returned, it is only one key at a time. This means that if you want to accept a series of characters as input, you must write the program to accept them one at a time and combine them within the program. Furthermore, nothing is displayed on the screen when the keyboard is read, so your program has to do this itself.

To make all these concepts clear and to set the stage for the assembly language program which comes later in this chapter, here is an Extended Basic program which does essentially the same things. It clears the screen, then prompts for and accepts a number, checking to make sure that each character typed is a digit (beeping if an invalid character is pressed).

```
100 CALL CLEAR
110 DISPLAY AT(13,14):"DATA:"
120 ACCEPT BEEP AT(13,19)VALIDATE(DIGIT):T
130 PRINT T
```

Here is a Basic program which does the same thing, but without the aid of commands such as CALL CLEAR, DISPLAY, and ACCEPT. This second Basic program is very similar to the way you have to do these things in assembly language.

```
100 FOR R=1 TO 24
110 FOR C=1 TO 32
120 CALL HCHAR(R,C,32)
130 NEXT C
140 NEXT R
150 CALL HCHAR(13,14,68)
160 CALL HCHAR(13,15,65)
170 CALL HCHAR(13,16,84)
180 CALL HCHAR(13,17,65)
190 CALL HCHAR(13,18,58)
200 C=19
210 N=0
220 CALL SOUND(100,1500,0)
230 CALL KEY(0,K,S)
240 IF S<1 THEN 230
250 IF K=13 THEN 330
```

```
260 K=K-48
270 IF K<0 THEN 350
280 IF K>9 THEN 350
290 N=N * 10 + K
300 CALL HCHAR(13,C,K+48)
310 C=C+1
320 GOTO 230
330 PRINT N
340 STOP
350 CALL SOUND(200,200,0)
360 GOTO 230
```

Statements 100 through 140 clear the screen, one character at a time. Statements 150 through 190 put "DATA:" on the screen, one character at a time. (In fact, this is somewhat different than assembly language, in which you can put more than one character on the screen, somewhat similar to DISPLAY in Extended Basic.) Statement 200 sets C to point to the screen position for input, and statement 210 sets N to 0 (to be used to accumulate the number as it is keyed in). Statement 220 generates the tone to prompt the user to begin data entry. The entry loop begins at 230, where KEY is called. If no key is being held down, statement 240 loops right back to 230. If the key pressed was the "ENTER" key, K is 13; statement 250 transfers to 330 to print the value of N and then stop. If not ENTER, subtract 48 from K; this maps the character "0" to the value 0, maps "1" to 1, and so forth. After doing this, if K is less than 0 as tested in statement 270, go to 350 because it was not a digit. Similarly, at 280, check to see if K is now greater than 9. If K is a digit from 0 to 9, multiply N by 10 and add K; this accumulates the number from the multiple key presses. At statement 300, display the character at position C on the screen, then add 1 to C to move to the right, and go back to wait for another key to be pressed. At 350, generate a tone to signify an improper key press, then go back to the top of the loop at 230.

On the next page is an assembly language program to do the same thing. The Basic program above was written with almost identical logic to facilitate comparison of Basic and assembly language doing the same things. However, one difference is that this program does not print the number once the ENTER key is pressed. Instead, the value is stored in location >7200.

The assembly language program is shown using the syntax of the Dow Editor/Assembler because it was take directly from Section 10 of the Dow Editor/Assembler manual.

```
   DEMO PROGRAM - INPUT AND DISPLAY
   INTEGER VALUE (LOAD AT 7500)
1  000      LI   R1;766  CLEAR SCREEN
   004      LI   R2;>8080 (BLANKS)
2  008 TOP:MOV   R2;@BUF(R1)
   00C      DECT R1
   00E      JOC  TOP
3  010      CLR  R0       WRITE
   012      LI   R1;BUF   BLANKS
   016      LI   R2;768   TO
   01A      BLWP @MBW     SCREEN.
4  01E      LI   R0;392   WRITE PROMPT
   022      LI   R1;PRO   TO
   026      LI   R2;5     SCREEN.
   02A      BLWP @MBW
5  02E      LI   R0;397   INPUT POS.
6  032      CLR  R2       R2=NUMBER.
7  034      MOVB R2;@MOD  MODE 0.
8  038      BLWP @GPL     ACCEPT
   03C      DATA >34      TONE.
9  03E LP :LIMI 2         ALLOW INTER-
   042      LIMI 0        RUPTS BRIEFLY.
10 046      BLWP @KEY     CALL KSCAN.
   04A      MOVB @STA;R1  CHECK STATUS
   04E      COC  @MSK;R1  FOR NEW KEY.
   052      JNE  LP       NOT YET.
11 054      CLR  R1       YES.
   056      MOVB R1;@STA  CLR STATUS.
12 05A      MOVB @INP;R1  LOOK AT IT.
   05E      SWPB R1
13 060      CI   R1;>D    ENTER?
   064      JEQ  END      GO IF DONE.
14 066      AI   R1;-48   NO. CHECK
   06A      JLT  ERR      FOR DIGIT.
   06C      CI   R1;9
   070      JGT  ERR
15 072      MPY  @V10;R2  OK. COMPUTE
   076      MOV  R3;R2    NUMBER
   078      A    R1;R2    IN R2.
16 07A      AI   R1;>90   NOW WRITE
   07E      SWPB R1       DIGIT TO
17 080      BLWP @SBW     SCREEN.
18 084      INC  R0
   086      JMP  LP       GO FOR NEXT.
19 088 END:MOV   R2;@>7200 STORE R2.
   08C      B    *R11     BACK TO BASIC.
20 08E ERR:BLWP @GPL      ERROR.
   092      DATA >36      BAD TONE.
   094      JMP  LP
   096 PRO:BTXT /DATA:/
   09C MSK:DATA >2000     MASK.
   09E V10:DATA 10        VALUE TEN.
```

```
0A0 MOD:EQU  >8374   MODE.
0A2 BUF:EQU  >7200   BUFFER.
0A4 KEY:EQU  >6020   KSCAN.
0A6 STA:EQU  >837C   STATUS.
0A8 INP:EQU  >8375   KEY PRESSED.
0AA GPL:EQU  >6018   GPLLNK.
0AC SBW:EQU  >6024   VSBW.
0AE MBW:EQU  >6028   VMBW.
```

At 1, the first nine instructions blank out the screen by filling VDP locations 0 through 767 with blanks. (If Basic is running, all characters must be biased by >60, so a blank is >80 instead of >20. This is discussed and demonstrated during the discussion of the BTXT directive in the chapter on directives.) The 768 blanks are first loaded into CPU RAM, starting at BUF; this is done with a loop at 2. At 3, they are written into VDP RAM by calling VMBR. This is relatively easy to use. First, load register 0 with the address in VDP where data is to be put; this corresponds to the screen position, and is 0 to blank out the entire screen. Second, load register 1 with the address in CPU memory of the data to be obtained; this value is the address of BUF in this instance. Last, load register 2 with the number of bytes to be sent (768 to blank the entire screen).

At 4, locations 01E to 02A write "DATA:" to the screen at position 392. As when blanking out the screen, register 0 holds the address in VDP memory where the data is to be sent; 392 for row 13 column 8. Register 1 holds the address in CPU memory of the data to be sent; in this case, the label PRO (for prompt) points to the string "DATA:". Finally, register 2 is loaded with the value 5, which is the number of characters to be sent to VDP memory with VMBW.

At 5, R0 is set to point to the input position, which is immediately to the right of the prompt message on the screen.

At 6, register 2 is cleared to accumulate the number to be entered.

At 7, the location identified by the symbol MOD is set to 0 by moving a byte from register 2. This location corresponds to the first argument in CALL KEY. It determines the mode when reading from the keyboard. Although there is a CLR instruction,

were to be used, CLR @MOD, it would clear two bytes. Therefore, it is necessary to have a byte value 0 someplace to be moved into MOD. Since register 2 has just been cleared (for an entirely different reason), it is handy to use it: MOVB R2;@MOD. When doing this, watch out so that the two instructions do not become separated. If they do, the wrong value may end up in register 2, causing unpredictable behavior on the part of KSCAN.

At 8, the prompt tone is started. This is done with the GPLLNK routine. It can do many things. You indicate which to do by the value in the next word, >34 in this case. This routine is described in the TI Editor/Assembler manual, starting at page 251, and in the Mini Memory manual, starting at page 38. Remember the discussion about interrupts; they should not be enabled when GPLLNK is called, but they have to enabled at intervals in order for the sound to function properly.

At 9 is the top of the input loop. First, interrupts are allowed briefly. This is necessary so a tone can be generated.

At 10, KSCAN is called. The status byte is moved to R1 so bit 2 can be tested. (The COC instruction uses MSK, which contains a mask value that identifies the single bit to be tested.) If the bit is set, a key has been pressed. If not set, loop back to LP.

At 11, the status byte is reset to 0. As discussed above, there is no clear byte instruction. Therefore, register 1 is cleared and then MOVB is used.

At 12, the byte holding the key that was pressed is moved from INP to register 1 with MOVB. The move puts the byte value into the left half of the register. Register 1 had just been cleared (for another reason); this means that after the SWPB R1, the character is in the right of R1 and the left is 0's. Now, at 13, it is an easy matter to check to see if the ENTER key, value >D, was the key that was pressed; go to END if so.

At 14, subtract 48 from the code for the key that was pressed, and if the result is less than 0, go to ERR. Then compare the register to 9, and go to ERR if it is greater.

At 15, the character has been tested to make sure it is a digit, so it is now time to add it into the total for the number being accumulated in register 2. Do this by multiplying the number so far by 10. This leaves the product in registers 2 and 3. However, the product is assumed to be small enough to fit into just one register, which is register 3. Therefore, register 3 is moved into register 2. The digit just keyed in can now be added into register 2 from register 1.

To show the digit on the screen so the user knows that it has been accepted, at 16 bias the digit by two values. First, if the program is called from Basic, the digit must be biased by >60 so it can be displayed. Second, because register 1 now contains the value (instead of the code for the character), the digit must be biased by >30 = 48 (because it was subtracted out above). In other words, add >90 to it. Then swap it to the left byte.

At 17, put the character on the screen using VSBW (single

byte write). For this routine, register 0 is supposed to hold the VDP address where the single byte is to be sent. Register 1 holds the byte in the left side.

At 18, increment the screen position pointer and loop for another digit.

At 19, terminate with B *R11. (If called from Basic, this will return properly.) The accumulated value is first stored at >7200.

It is not safe to return to Basic if the STATUS byte is not first cleared. In this case, it was cleared after the last KSCAN.

At 20, the bad response tone is generated using GPLLNK in a manner very similar to the generation of the prompt tone.

Finally, data and equates end the program. As discussed above, the BTXT directive is used if the program is called from Basic.

APPENDIX A: CPU MEMORY MAP

This appendix lists in numerical order many of the addresses of interest in the CPU memory. The data was derived from the TI Editor/Assembler manual and no claim is made for its accuracy or completeness.

Locations 0000-1FFF

The first 8K (0000-1FFF) is console ROM.
General page references are: 398, 399, 400, and 401.

LOCATIONS	LABEL	DESCRIPTION EDITOR/ASSEMBLER MANUAL PAGE REFERENCES
0000,1		Return to color bar screen. Pages: 442
000E,F	SCAN	Keyboard scan routine Pages: 247,415
0010,1		Return to calling program. Pages: 442

Locations 2000-3FFF

The next 8K (2000-3FFF) is low memory expansion.
General page references are 398, 399, and 400.

LOCATIONS	LABEL	DESCRIPTION EDITOR/ASSEMBLER MANUAL PAGE REFERENCES
2000-????		E/A loader and assembly language programs, XB utilities and assembly language programs. Pages: 305,410
2000,1		ID code >A55A for E/A loader Pages: 411
2000,1		XML vector for XB Pages: 412
2002-????		XML vectors for E/A Pages: 411
2002-????		Utility data area for XB Pages: 412
2006,7		ID code >AA55 for XB Pages: 412
2008-3FFF		Utility BLWP vectors, routines, assembly language programs, DEF table Pages: 412
2008,B		NUMASG in XB Pages: 416
200A-2019		Argument identifiers used by GPLLNK in Basic Pages: 278
200C,F		NUMREF in XB Pages: 416
2022-????	UTLTAB	Utility table entry address Pages: 247,263,308,411
2024,5	FSTHI	First free address in high memory Pages: 265,305
2026,7	LSTHI	Last free address in high memory Pages: 265
2028,9	FSTLOW	First free address in low memory Pages: 265,276,305

APPENDIX A: CPU MEMORY MAP

202A,B	LSTLOW	Last free address in low memory Pages: 265,276,307
202C,D	CHKSAV	Check sum Pages: 265
202E,F	FLGPTR	Pointer to the flag in the PAB. Pages: 265
2030,1	SVGPRT	GPL return address. Pages: 265
2032,3	SAVCRU	CRU address in the peripheral. Pages: 265
2034,5	SAVENT	Entry address of the DSR or subprogram. Pages: 265
2036,7	SAVLEN	Device or subprogram name length. Pages: 265
2038,9	SAVPAB	Ptr to the device or subprogram name in PAB. Pages: 265
203A,B	SAVVER	Version number of the DSR. Pages: 265
2094-20C3	UTILWS	Utility workspace registers Pages: 246
20BA-20DB	USRWSP	User workspace registers Pages: 246,440,441
2100-????		Utility BLWP vectors for E/A Pages: 411
2128-????		Utility routines for E/A Pages: 411
2676-3F7F		Expansion area for relocatable programs. Pages: 305
2700-????		Relocatable assembly language programs, E/A Pages: 411
3F38-3FFF		REF/DEF table Pages: 246,308,411

Locations 4000-7FFF

The next 8K (4000-5FFF) is reserved for ROM in peripherals.
General page references are 399, 400, and 401.

The next 8K (6000-7FFF) is reserved for ROM in modules.
General page references are 399, 400, and 401.

Locations 8000-9FFF

The next 8K (8000-9FFF) is console RAM and memory mapped addresses. General page reference: 399.

LOCATIONS	LABEL	DESCRIPTION EDITOR/ASSEMBLER MANUAL PAGE REFERENCES
8300-83FF	PAD	Scratch pad (THIS IS THE ONLY RAM) Pages: 247,308,415
8300-8340		Destroyed by GPLLNK 003B Pages: 253
8300-8317		Not used by Basic if assembly language routine has no parameters. Pages: 404
8318-8349		Used by Basic Pages: 404
830C,D		No. of types to be allocated by GPLLNK 0038 Pages: 253
8310,1		Value stack pointer used by LINK in Basic. Pages: 278
8312		No. arguments in LINK in Basic. Pages: 278
831A,B		Pointer to first free address in VDP RAM Pages: 253
831C,D		Pointer to allocated string space Pages: 253
831C,D		Pointer to PAB for ERR Pages: 287
8328-????		Convenient area for speech read routine. Pages: 349,350
834A-8351	FAC	Floating point accumulator Pages: 252,285,286,290,415
834A-836D		Used by DSR's. Pages: 300,404
8354		Error code returned by mathematical routines Pages: 254,259,261

8354,5	????	No. of chars in cass name for GPLLNK 003D Pages: 253
8356,7		Ptr to char past cass name for GPLLNK 003D Pages: 253
8356,7		Ptr to name len in PAB for DSRLNK or LOADER Pages: 262,263
8356,9		Used by GPLLNK 0038 Pages: 253
835C-????	ARG	Arguments for GPL routines Pages: 252
836D		Set to >08 for GPLLNK 0038 Pages: 253
836E,F	VSPTR	Ptr into VDP RAM for Value Stack ROM math Pages: 252,259,260,404
8370,1		Highest available VDP RAM address Pages: 252,404
8372		Least significant byte of value stack ptr. Pages: 404
8373		Least significant byte of subr stack ptr. Pages: 404
8374		Selects keyboard device Pages: 250,404
8375,6		Used by some mathematical routines Pages: 255,256
8375		ASCII value of key pressed Pages: 251,405
8376		Joystick Y-position Pages: 250,405
8377		Joystick X-position Pages: 250,405
8378		Random number generator Pages: 405
8379		VDP interrupt timer Pages: 405

837A		No. sprites in motion. Pages: 340,347,405
837B		VDP status byte Pages: 405
837C	STATUS	STATUS byte Pages: 250,298,299,311,405,410,441
837D		VDP character buffer Pages: 405
837E		VDP current row Pages: 405
837F		VDP current column Pages: 405
8380-839F		Default subroutine stack Pages: 405
8386,7		Highest loc available for ass'y lan with XB Pages: 410,412
8388,9		Used by Basic Pages: 405
838A-83BF		Subroutine and data stack areas for Basic Pages: 405
83A0-83BF		Default data stack Pages: 405
83C0,1		Random number seed Pages: 405
83C0-83DF		Interpreter workspace Pages: 405
83C2		Flag byte to control interrupt routine. Pages: 406
83C4,5		Address of user defined interrupt routine. Pages: 406
83CA		Console keyboard debounce. Pages: 406
83CC,D		Sound list pointer in VDP RAM. Pages: 312,321,322,323,405

83CE		Set to >01 to start sound generator Pages: 312,321,322,323,406
83D0,1		Should be set 0 before GPLLNK 003D Pages: 253,406
83D4		Should be copy of VDP Register 1 Pages: 248,326,406
83D6		Screen time out counter Pages: 406
83D8,9		Return address for scan routine. Pages: 406
83DA		Player number for scan Pages: 406
83DA		Used by DSR if not interrupt driven. Pages: 300
83E0,F	GPLWS	GPL workspace Pages: 247,308,406,415
83F0		Set LSB for sound Pages: 312,321,322,323
8400	SOUND	Sound chip register Pages: 308,317,415
8800	VDPRD	VDP RAM read data Pages: 247,267,308,402,415
8802	VDPSTA	VDP RAM status Pages: 247,269,308,402,415
8C00	VDPWD	VDP RAM write data Pages: 247,268,308,402,415
8C02	VDPWA	VDP RAM write address Pages: 247,402,415
9000	SPCHRD	Speech read data register Pages: 308,351,415
9400	SPCHWT	Speech write data register Pages: 308,351,415
9800	GRMRD	GROM/GRAM read data Pages: 247,271,308,415

APPENDIX A: CPU MEMORY MAP

9802	GRMRA	GROM/GRAM read address Pages: 247,270,308,415
9C00	GRMWD	GROM/GRAM write data Pages: 247,271,308,415
9C02	GRMWA	GROM/GRAM write address Pages: 247,270,308,415

Locations A000-FFFF

The last 24K (A000 - FFFF) is high memory expansion.
(Extended Basic stores numeric data and the program here.)
General page references:399, 410.

LOCATIONS	LABEL	DESCRIPTION EDITOR/ASSEMBLER MANUAL PAGE REFERENCES
A000-FFD7		Relocatable assembly language programs, E/A. Pages: 305,410,411
A000-FFE0		Free space (see >8386), numeric values, line number table, XB program. Pages: 412
FF08-FFFF		Used by XOP1 Pages: 400

APPENDIX B: VISUAL DISPLAY PROCESSOR MEMORY MAP

This appendix lists in numerical order many of the addresses of interest in the CPU memory. The data was derived from the TI Editor/Assembler manual and no claim is made for its accuracy or completeness. In particular, there are many ways VDP memory can be used, so the table below is only representative.

Visual Display Processor Memory Map

LOCATIONS	DESCRIPTION/REFERENCES
0000-02FF	Screen image table for E/A Pages: 330,343,344,403
0300-037F	Sprite attribute list for E/A Pages:339,347,403
0380-039F	Color table for E/A Pages:330,342,347,403
0400-077F	Sprite descriptor table (characters 80-EF) Pages:339,347,403
0780-07FF	Sprite motion table (4 bytes each sprite) Pages:340,347,403
0800-0FFF	Pattern Descriptor Table for E/A Pages:329,403
1000-37D6	PAB's, buffers, free space for E/A Pages:403
1800	Usual screen image table for bit map mode Pages:334
37D7-3FFF	Blocks reserved for diskette DSR Pages:403

APPENDIX C: DECIMAL TO HEXADECIMAL CONVERSION PROGRAM

This appendix lists a Basic program which can be used to convert from decimal to hexadecimal notation, or from hexadecimal to decimal notation.

```
100 REM  (HEXDEC AND DECHEX CONVERSION)
110 INPUT A$
120 IF SEG$(A$,1,1)=">" THEN 250
130 REM  DECHEX - DECIMAL TO HEX
140 HEX$=""
150 N=VAL(A$)
160 IF N>=0 THEN 180
170 N=65536+N
180 T=INT(N/16)
190 D=N-16*T
200 HEX$=SEG$("0123456789ABCDEF",D+1,1)&HEX$
210 N=T
220 IF N>0 THEN 180
230 PRINT "  >";HEX$
240 GOTO 110
250 REM  HEXDEC - HEX TO DECIMAL CONVERSION
260 N=0
270 FOR I=2 TO LEN(A$)
280 D$=SEG$(A$,I,1)
290 D=POS("0123456789ABCDEF",D$,1)-1
300 N=N * 16 + D
310 NEXT I
320 PRINT " ";N,N-65536
330 GOTO 110
```

Enter hexadecimal values with the ">"; example: >FFFF. The value is displayed in decimal. If negative, it is displayed also in the equivalent value assuming the sign bit is the value 32,768. Example: >FFFF is both -1 and 65,535.

Enter positive or negative decimal values without the ">"; example: 5230. The result is displayed in hex with the ">".