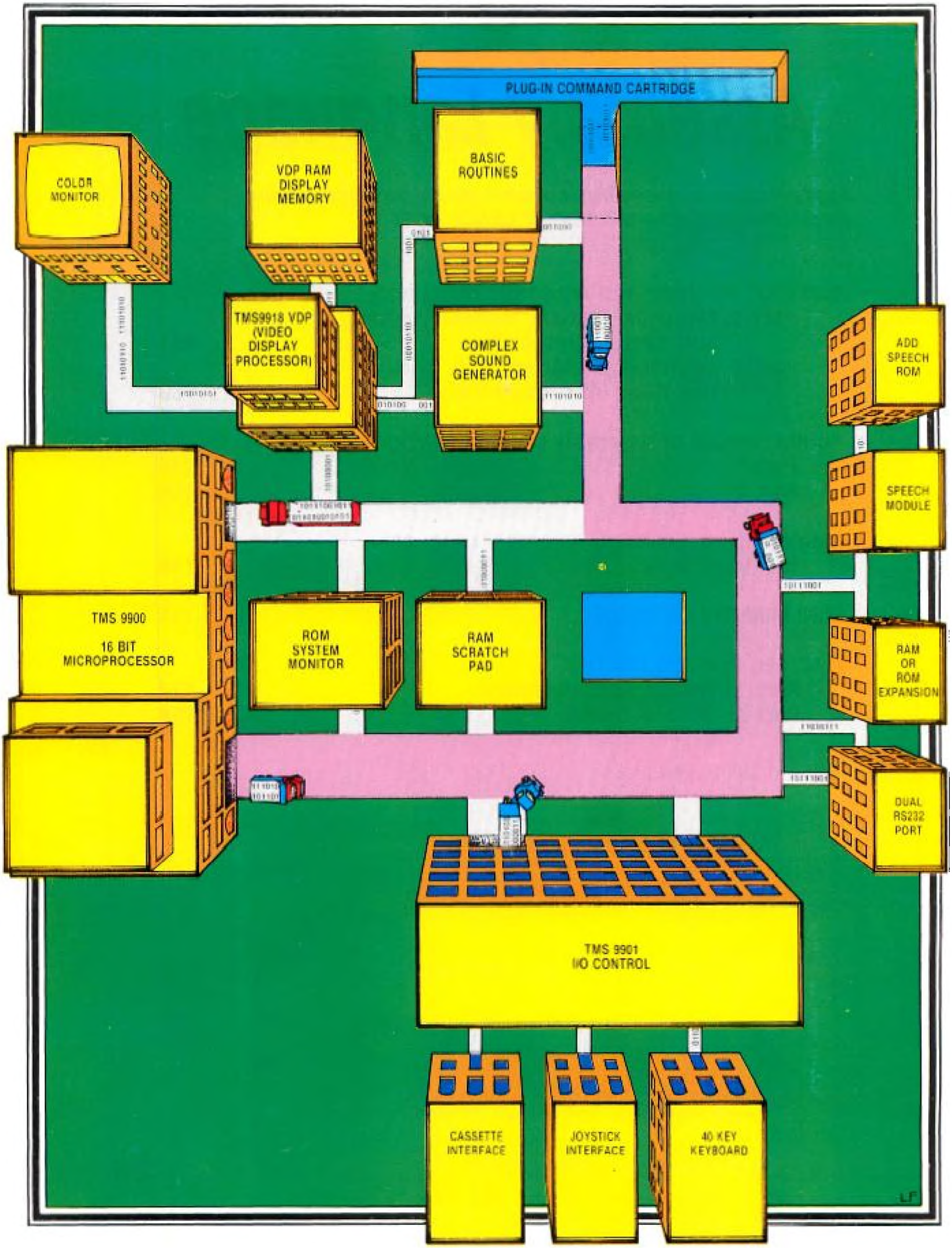
5

Assembly Language



5

Assembly Language

Faster than a speeding cursor! More powerful than Extended BASIC!—It's SUPER LANGUAGE!

TMS9900 Machine and Assembly Language:

Part 1: Electrical Signals, Number Systems and CPU Architecture 131
Part 2: Registers, Programming, and the Need for Assemblers 133

Fundamentals of Assembly Language Programming:

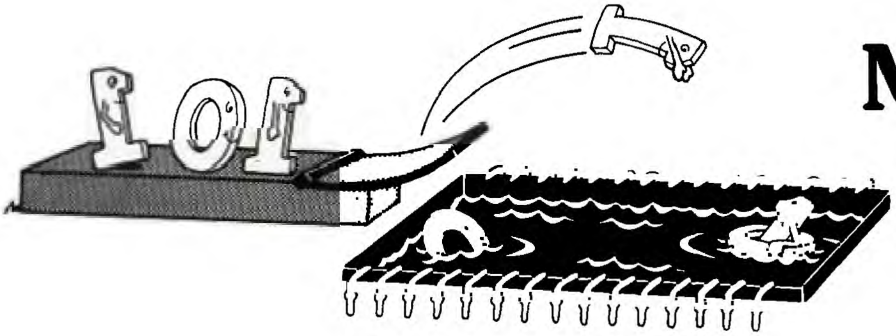
Part 1 136
Part 2 138

Magic Crayon: Learning Assembly Language the Hard Way 146

MINI MEMORY Cartridge 154

A Screen Printing Utility:

Part 1: Design Considerations 157
Part 2: Screen Dump 158



TMS9900 Machine & Assembly Language

PART 1: Electrical Signals, Number Systems & CPU Architecture

If you're a reader of *99'er Home Computer Magazine*, you are probably aware that there is a difference between 8-bit and 16-bit computers . . . although just exactly *what* that difference is—other than "16 bits are twice as many as 8 bits"—might not be that obvious. My purpose in this series of articles is, therefore, to discuss the inner workings of your 16-bit computer by gradually introducing you to its operation and low-level programming in a language much closer to the way your computer operates without any BASIC interpreter slowing things down, or coming between you and the power of your machine.

The heart of any computer is its microprocessor, and the one we'll be examining is, naturally enough, the Texas Instruments TMS9900—the 16-bit chip around which this magazine is organized. To understand its operation, we first have to know something about electrical signals and number systems, so let's begin our discussion here.

Clocks, Pulses, Bits & Bytes

The electrical signals used by a computer are labeled high and low, or 1 and 0, respectively. One of these signals is called a *bit*. Inside the computer this corresponds to one wire. All of the wires together are called a *bus*. The computer reads and writes a part of the bus called the *data bus* at specific intervals, which are regulated by a *clock*. The signals that the clock produces to tell the computer when to read and write are called *clock pulses*.

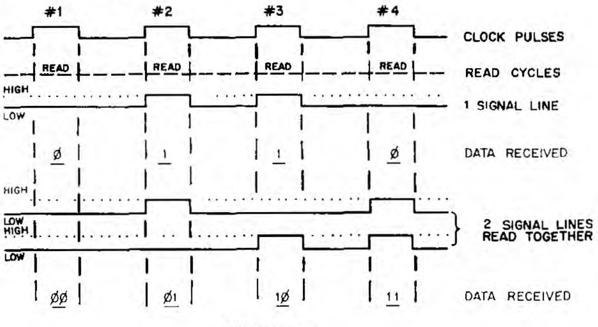
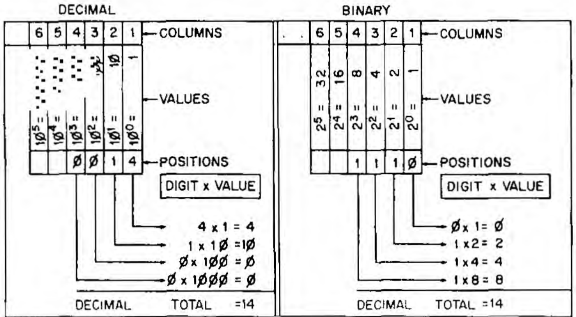


Chart 1

At each pulse of the clock, the computer reads a group of lines. Your normal, run-of-the-mill microcomputer uses groups of 4,8,or 16 bits. All the information read or written is called *data*. If the computer is reading or writing on 1 line, the data is called *serial*. If it is reading or writing on a group of lines together, the data is called *parallel*. 4 bits in parallel are called a *nybble*; 8 are a *byte*; and 16 has no name, but I propose to call it a *gobbyl*.

Look at Chart 1. The top line is the clock. In this example when the pulse is high, the computer reads the signal lines. Notice that when there is only one signal line, the data received can be only a 1 (when the line is high) or a 0 (when the line is low). There are only two possible codes you could see during one clock pulse. You would see a 1 or a 0.

Now look at what happens when you have two signal lines grouped together: 4 different codes are possible. On clock pulse #1 both lines are low (code 00); on pulse #2 the bottom line is low and the top one is high (code 01); pulse #3 has the bottom high and the top low (code10); and pulse #4 has both lines high (code 11).



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

Chart 2

Number Systems

These codes could also be considered numbers. Counting with only 0's and 1's is called *binary* (from the Latin word for two) or *base two* counting. Ordinary, plain, vanilla numbers that we use everyday that are called *decimal* (from the Latin for ten, of course) or *base ten* numbers. Even though we have only the ten digits from 0 to 9, we can make very large numbers by using the same digits in different positions. Follow along on chart 2.

The position on the extreme right in a decimal number is the ones column. For that matter, the position on the extreme right in any base is the ones column. Why? Because you find the column value by taking the number of digits you have and raising it to the power of column minus one. For example, if you have ten digits, and the column is number 1 (from the right), then the value of that column is 10 to the 1 minus 1, or 10 to the 0 power. Any number to the 0 power is 1, so the first column is always ones in any base.

The second column is a different matter. In base ten it is 10 to the 2 minus 1, or 10 to the 1st power, or 10. So if you write 14 what you mean is 4 groups of ones and 1 group of tens. In base two the second column (from the right) would be 2 to the 2 minus 1, or 2 to the 1st, or 2. The second column or position in binary is the twos column.

The thing that makes the zero so neat is that it holds the position without giving it a value. Zero ones is zero! If you just left a blank there, people would have to write all their numbers in little boxes or pretty soon the columns would get all jumbled up. Is there one blank or two? . . .or three?? Better use the zero.

The columns in binary numbers are just like the signal lines in a computer. In theory, the columns go on forever—and so do the numbers. Regardless of the base you are in, you can keep writing numbers forever! But wait! I just said that signal lines are usually groups of 4, 8, or 16. If signal lines are the same as columns, then there is a limit to the size of number a computer can understand. How big is the biggest number you can use?

To find out, raise the base to the same power as the number of positions you have. On chart 1 when we used two lines, that was 2 to the 2nd power, or 4 codes or numbers. With 4 lines, there are 16 (2 to the 4th); with 8 there are 256; and with 16 lines there are 65536.

The last code on the chart is 1111, which in decimal is 15. I said you could get 16 numbers with four lines, so where is the last number? Don't forget to count 0! 0 through 15 is sixteen numbers, 0 through 255 is 256 numbers, and so on.

There are other bases, or course. The numbers marked *hexadecimal* are from a base with 16 digits—the normal 10 digits from 0 to 9, plus the letters A to F. Use them just like any other digits. For instance, on the chart, 1111 binary is 15 decimal and F in hexadecimal (hex for short). The next number in hex is 10; in decimal it is 16; and in binary you have to add a new position (sixteens) and write 10000.

You can always add as many zeros to the front of a number as you want without changing it. However, if you make a binary number divisible into groups of four, an interesting thing happens: Each group of four can represent 16 codes or numbers. Since that is exactly the number of digits in the hex number system, you can substitute! This makes long binary numbers much easier to read, and doesn't change their values at all.

Try a few yourself. They're easy!

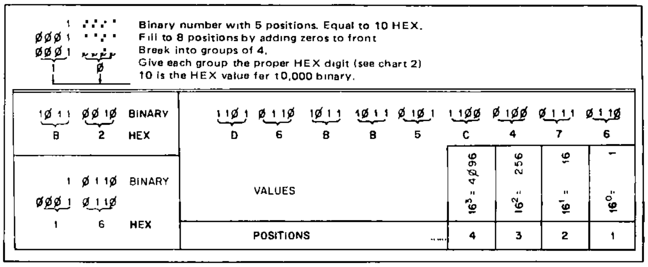

Binary number with 5 positions. Equal to 10 HEX.
Fill to 8 positions by adding zeros to front
Break into groups of 4.
Give each group the proper HEX digit (see chart 2)
10 is the HEX value for 10,000 binary.

Chart 3

Hardware

The TMS9900 is called a 16-bit *CPU (Central Processing Unit).* This means that when it fetches an instruction from memory, it gets 16 bits in parallel. And when it reads or writes data this is usually done in groups of 16 bits too. [In the TI-99/4A, however, this 16-bit group is converted into an 8-bit data bus.—Ed.] You may hear the term *word* used for 16 bits. If you are talking about a 16-bit machine, the term is correct. But remember, if you are talking about an 8-bit CPU, 8 bits (or byte) is a word; if the CPU is 32 bits, the word is 32 bits.

It is necessary for a programmer to know about only two kinds of memory. *Random-access memory (RAM),* sometimes called *read/write memory,* is what stores the user's program, data, etc. The user or the computer can read or write in it. The memory location is chosen by the lines on the bus called *address lines.* The data that is being read or written appears on the data bus.

Read-only memory (ROM) comes in many varieties and works just like RAM except for one thing—it can't be written to. If you tell the computer to write, it will go through the motions of writing, but it doesn't work. The old data is still there.

Inside the CPU there are a few memory locations that are not addressed by the address bus. The chip itself knows where they are. These are called *registers.* All *machine language* and *assembly language* programming involves manipulating the data in these registers, because that is all that the computer really can do!

How many registers there are and how big they are varies widely. The chip manufacturer usually labels the registers and decides on a short code, called an *operation code* (op-code), for each of the manipulations that the chip can do. An *assembler* is a program that reads these op-codes and writes them into memory in the binary form that the CPU understands. When you write a program using the op-codes, you are writing in *assembly language.* If you write your own assembler you can devise your own op-codes. But because the manufacturer generally writes an assembler for his chip, you can use his op-codes.

About the only thing all CPUs have in common is a register called the *Program Counter* (PC). The address bus is just an extension of the PC. Each bit of the program counter is, in effect, connected to one signal line of the address bus. Since the TMS9900 chip was designed especially for dedicated control purposes (e.g., production lines inspection or phone switching) where the pro-

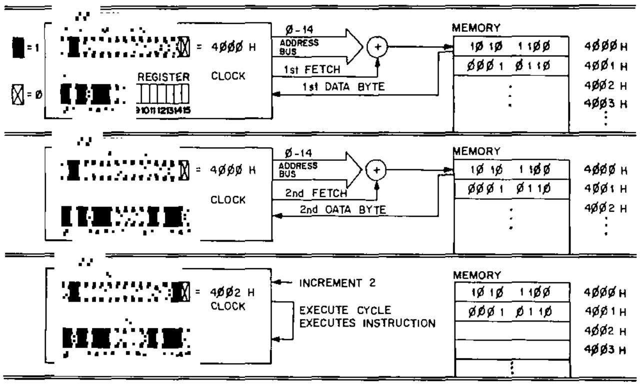


Chart 4

gram is always in ROM—and since at the time most ROMs were made for 8-bit computers—the address bus of the 9900 is a little unusual.

The bits of the PC allow the chip to address 65536 blocks of memory. The blocks could be any size, but as I said, most ROMs were in blocks of 8 because most computers had an 8-bit data bus. The PC in the 9900 has 16 bits. These are labeled 0-15, from (left to right), *most significant bit (MSB)* to *least significant bit (LSB)*. Why are there only 15 address lines? Follow on Chart 4 as we go along.

Normally the PC advances after each instruction or parameter it fetches so that it points to the next memory byte. But the 9900 needs 16 bits instead of the 8 available at each location in most ROMs. So the 9900 has two different fetch cycles: it reads the byte indicated by the PC on the first cycle, hooks the next byte to it on the second cycle, then increments the PC by two. To the user this all appears as one fetch, except that the PC is incremented by *two* instead of by *one* as expected. By eliminating the last bit, however, the address line appears to step normally. The drawback is that you can address only 32767 words. It's still 65536 bytes though.

PART 2: Registers, Programming & The Need For Assemblers

Status Register

Almost every CPU has some kind of *flag*(s). These are set (high) and reset (low) by actions performed in the manipulations of data. Different instructions affect different flags. Modern CPUs combine several flags into a single *Status Register*. The TMS9900 is no exception. Its Status Register (ST) is 16 bits long. Bits 7-11 are not used at present. The others are shown in the drawing below and are explained in the text.

TMS9900 STATUS REGISTER

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
L >	A >	EQ =	C	OV	P	X XOP									
			CARRY	OVERFLOW	PARITY		UNUSED					INTERRUPT MASK			

Each of these conditions will be discussed in more detail as examples are shown. Until then, these simple descriptions will help.

The four bits labeled 12-15 can select up to 16 interrupt levels. All levels equal to or above the level indicated are enabled.

Bit 0 is set after any operation where the destination value (answer) is greater than the source (the first operand used; it remains unchanged). All 16 bits are used for the comparison.

Bit 1 is similar to bit 0 except that the values are compared as signed integers. The MSB (most significant bit) designates the sign of the integer, with a 1 meaning negative and a 0 meaning positive. The range is +32,767 to −32,768. Negative numbers are represented in a two's complement fashion.

Computer math is cyclic. This means that if you add 1 to the highest possible 16-bit number (FFFF hex), you go back to 0000 hex with a *carry* bit that is set. If you subtract 1 from 0000 hex without the carry, you get an *overflow*; but if the carry is set, you get FFFF hex. Therefore, −1 is FFFF hex in two's complement. To see its usefulness, let's add −1 and 1: FFFF hex plus 0001 hex equal 0000, the carry is set, and the answer is zero. In a nutshell, this whole business of two's complements and carry bits is simply a way to subtract by adding.

Bit 2 is set if the two operands are equal.

Bit 3 is set if a 1 is shifted out of an operand, or if a carry occurs in a math operation.

Bit 4 is set if the math requested cannot be done.

Bit 5 is set if the parity is odd, and reset if it is even. Odd parity means that there is an odd number of 1s in the binary representation of an operand.

Bit 6 is set after an *extended operation* has been completed. This is done because an interrupt is not checked for after completion of an extended operation. (You therefore may wish to have the software check for one if this flag is set).

The ALU

Most CPUs have an *Arithmetic/Logic Unit (ALU)* where the simple math is performed. An *accumulator*, a special register used by the ALU, usually contains the answers to the math. In the TMS9900 there is no accumulator because the destination address serves as the equivalent of an accumulator. This means, in effect, that any memory location *can* be the accumulator. There is an ALU on the TMS9900 chip, but its operation is intrinsic to the instructions.

Other Registers

Most CPUs have a few extra registers where quickly-needed values can be stored, as well as a register called a *Stack Pointer* which points to a section of memory where more data can be "piled" and then quickly accessed. These two concepts have been combined on the TMS9900 into a single *Workspace Pointer Register (WP).* The WP points to a block of 32 bytes of the memory arranged as 16 workspaces (WS), each 16 bits long. The workspaces are synonymous with registers, and are used the same way. We can change the WP in several ways and can save the old WP when a new one is used. This allows us to return to the old one if we need to. This set-up, in effect, acts like an elaborate *stack.*

There are five different ways to use these WP registers to indicate an operand for an instruction. These *addressing modes* are as follows:

1. Workspace Register Mode code 00	—the data in the indicated register is the data used.
2. Workspace Register Indirect code 01	—the data in the register is treated as the address of the real data.
3. WS Register Indirect w/Auto-Increment code 11	—same as above, but the register is incremented upon completion.
4. Symbolic or Direct code 10	—the address of the data follows the instruction in memory.
5. Indexed code 10 Td or Ts equal 1-15	—same as above, but the value in the index register is added to the address.

There are three other addressing modes not dealing with registers *per se:* (1) The *immediate mode* has the data immediately follow the instruction code. In other words, the address of the data is the address immediately following the PC. (2) The *CRU mode* has the address of an external input/output (I/O) device determined by bytes 3-12 of register 12. (3) The *JMP instruction* (and all variations thereof) uses the last 8 bits of the instruction to determine where on a 256 byte page to jump. The PC indicates the center of the page, so the jump can be from PC – 128 to PC + 127. One byte is taken up by the jump instruction itself. The 8 bits store the relative jump in two's complement form.

Programming and the Need for Assemblers

If your CPU is the TMS9900, the simplest computer you could construct would be composed of a clock, a CPU, some memory, a few control switches, 16 data switches, 16 lights for read out, and 15 address switches. It would be crude and slow to program, but once programmed, it would operate as well as any other computer. But how could we program it?

Suppose we wanted to load register 1 with zero, and then increment it until its contents were equal to either 1024 (decimal) or the contents of register 2. The first step can be done several ways. Immediately loading register 1 with 0 comes to mind first. A little investigation of the instructions for the chip show that we could save a word of memory by using the *Clear* command. Figure 1 shows the register format for the various commands, and Figure 2 shows the *op codes* for the instructions.

Using this information, we can now determine the binary values of each word. Load Immediate uses the first 10 bits as the op code; the 11th bit is not used; and bits 12-15 select the register. This means the first word is

00000010000X0001, where X can be 1 or 0.

The second word is the value to load, and in this case would be all zeros.

Using our simplified computer, just flip each switch on if there is a 1 at the corresponding bit, off if there is a zero. Press the *Input* control switch (it might be called *Load*, or . . .), and the instruction is stored in whatever address the address switches are set to. Then add 1 to the address switches (which adds 2 to the PC) and set all the data switches to zero. Press *Input* again, and our complete instruction is ready.

FORMAT	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	OP CODE			B	T_d		D				T_s		S			
2	OP CODE								RELATIVE JUMP							
3	OP CODE						D				T_s		S			
4	OP CODE						C				T_s		S			
5	OP CODE								C				W			
6	OP CODE										T_s		S			
7	OP CODE											N				
8	OP CODE											N	W			
	IMMEDIATE VALUE															
9	OP CODE						D				T_s		S			

KEY

B 1=byte 0=word
T_d destination address mode
D destination address
T_s source address mode
S source address
C counter
W register number
N unused
RELATIVE JUMP from +127 to –128

T_d/T_s FIELD CODES

00 Register
01 Indirect
10 with R0, symbolic
10 with R1–R15, indexed
11 Indirect with increment

Figure 1.

If instead, we use the *Clear* instruction, we would use the single-operand general format with the first 10 bits being the op code. The next two bits indicate address mode, and the last 4 bits select the register. Since we want to clear the register itself (not the word it points to), the code is 00,and the whole instruction is 0000010011000001.

Even with a hex keypad and a small monitor program, it would be a very time-consuming process to piece together the binary words, and then convert to hex and type them in. Typing in 04C1 is easier than setting switches to

0000010011000001,

but putting together those op codes is just the tedious, boring kind of work that computers are supposed to free us of. So why not use them for that?

Why not, indeed. . .That's exactly what we'll do when we look at a TMS9900 assembler.

Figure 2.

Mnemonic	Op Code	Format	Status	Bits Affected	Meaning
A	1010	1	0–4		Add words
AB	1011	1	0–5		Add bytes
ABS	0000011101	6	0–4		Absolute Value
AI	00000010001	8	0–4		Add immediate
ANDI	00000010010	8	0–2		And immediate
B	0000010001	6	-- -- --		Branch
BL	0000011010	6	-- -- --		Branch and Link (R11)
BLWP	0000010000	6	-- -- --		Branch, load WP
C	1000	1	0–2		Compare words
CB	1001	1	0–2, 5		Compare byte
CI	00000010100	8	0–2		Compare immediate
CKOF	0000001111000000	7	-- -- --		External Control
CKON	0000001110100000	7	-- -- --		External Control
CLR	0000010011	6	-- -- --		Clear
COC	001000	3	2		Compare Ones Corresp. (OR)
CZC	001001	3	2		Compare Zero Corresp. (AND)
DEC	0000011000	6	0–4		Decrement by one
DECT	0000011001	6	0–4		Decrement by two
DIV	001111	9	4		Divide
IDLE	0000001101000000	7	-- -- --		Computer idles
INC	0000010110	6	0–4		Increment by one
INCT	0000010111	6	0–4		Increment by two
INV	0000010101	6	0–2		Invert (complement)
JEQ	00010011	2	-- -- --	(ST2=1)	Jump if equal
JGT	00010101	2	-- -- --	(ST1=1)	Jump greater than
JH	00011011	2	-- -- --	(ST0 and ST2=1)	Jump high
JHE	00010100	2	-- -- --	(ST0 or ST2=1)	Jump high or equal
JL	00011010	2	-- -- --	(ST0 and ST2=0)	Jump low
JLE	00010010	2	-- -- --	(ST0=0 or ST2=1	Jump low or equal
JLT	00010001	2	-- -- --	(ST1 and ST2=0)	Jump less then
JMP	00010000	2	-- -- --	(none checked)	Jump unconditionally
JNC	00010111	2	-- -- --	(ST3=0)	Jump no carry
JNE	00010110	2	-- -- --	(ST2=0)	Jump not equal
JNO	00011001	2	-- -- --	(ST4=0)	Jump no overflow
JOC	00011000	2	-- -- --	(ST3=1)	Jump on carry
JOP	00011100	2	-- -- --	(ST5=1)	Jump odd parity
LDCR	001100	4	0–2, 5		Load CRU
LI	00000010000	8	0–2		Load immediate
LIMI	00000011000	8	12–15		Load immed. INT mask
LREX	0000001111100000	7	12–15		External control
LWPI	00000010111	8	-- -- --		Load immed. WP
MOV	1100	1	0–2		Move word
MOVB	1101	1	0–2, 5		Move byte
MPY	001110	9	-- -- --		Multiply
NEG	0000010100	6	0–4		Negate (2's comp.)
ORI	00000010011	8	0–2		OR immediate
RSET	0000001101100000	7	12–15		External control
RTWP	0000001110000000	7	0–6, 12–15		Return with WP
S	0110	1	0–4		Subtract word
SB	0111	1	0–5		Subtract byte
SBO	00011101	2	-- -- --		Set CRU bit to one
SBZ	00011110	2	-- -- --		Set CRU bit to zero
SETO	0000011100	6	-- -- --		Set ones
SLA	00001010	5	0–4		Shift left (0 fill)
SOC	1110	1	0–2	Words (OR)	Set ones corresp.
SOCB	1111	1	0–2, 5	Bytes (OR)	Set ones corresp.
SRA	00001000	5	0–3		Shift right (MSB fill)
SRC	00001011	5	0–3		Shift right circular
SRL	00001001	5	0–3		Shift right zero fill
STCR	001101	4	0–2, 5		Store from CRU
STST	00000010110	8	-- -- --		Store ST
STWP	00000010101	8	-- -- --		Store WP
SWPB	0000011011	6	-- -- --		Swap bytes
SZC	0100	1	0–2	Words (AND)	Set zero corresp.
SZCB	0101	1	0–2, 5	Byte (AND)	Set zero corresp.
TB	00011111	2	2		Test CRU bit
X	0000010010	6	-- -- --		Execute
XOP	001011	9	6		Extended operation
XOR	001010	3	0–2		Exclusive OR

IT'S SUPER LANGUAGE

PART 1: Fundamentals of Assembly Language Programming on the TI-99/4A

Before getting into the details of the TI-99/4A Editor/Assembler package, we should first consider what an assembler is and what it can do for us. Most readers are already familiar with the TI BASIC language, and many have already experienced the disk-oriented features of Extended BASIC. These BASICs are *interpreted* languages. When a BASIC program is being run, the BASIC interpreter converts (interprets) the BASIC statements, one statement at a time, into *machine language*—the binary ones and zeros that the computer understands. It then executes the statement it has just converted. Since a single BASIC statement usually generates several machine instructions, programs can execute relatively slowly. This is especially true in programs containing loops because each statement in a loop is interpreted each time it is encountered.

BASIC programs are simply input and RUN, but programming in assembly language involves an extra step which is not apparent in BASIC programming—namely the *assembler stage*: Assembly language programs must be input, then assembled and finally RUN. The assembler converts the assembly language statements (or *source* program) to machine language; it is the machine-language (or *object*) program which is RUN. Because there is no waiting for each statement to be interpreted at runtime, programs written in assembly language run extremely fast.

Another major difference between BASIC and assembly language is the difficulty of writing programs. A BASIC program is relatively easy to code because the instructions are English-like and the programmer does not have to worry about where variables reside in memory or have to understand the structure of the machine. Assembly language programs, on the other hand, are harder and more time-consuming to write because the instructions are machine-oriented (see "TMS9900 Machine and Assembly Language") and the programmer must understand the structure of the machine. Debugging assembly language programs is harder, too. But these difficulties are not necessarily disadvantages, because an understanding of the machine allows a programmer to create more efficient programs. Programming in assembly language is an education in itself, and is one of the best ways to learn how a computer works.

A programmer must consider these tradeoffs in choosing the best language for each application. In general, BASIC is faster to write and debug, but assembly language programs execute faster. Happily, TI has made it possible to choose *both* by enabling Extended BASIC programs to CALL assembly language subroutines. This means that a programmer can write mainly in Extended BASIC and use assembly language for portions of the program where faster execution is required (loops, and especially, sorts). Writing short assembly language subroutines to CALL from Extended BASIC programs is a good way to ease into assembly language programming, and after some practice you may find yourself writing entire applications in assembly language.

What follows is a preliminary look at the TI-99/4A Editor/Assembler package. It is, however, only an *overview* of the product. Other sections will go into more depth on specific features of the software.

Software Media and Required Hardware

The Editor/Assembler software resides in a Command Cartridge and on a disk. To run it, you'll need at least one disk drive and the 32K expansion RAM. Both the Editor and the Assembler are selectable from menus, and most of the screens include easy-to-understand prompting messages.

The Editor

The Editor is used to input Assembly Language source programs initially, to update programs previously saved on disk and to print programs. The Editor's features compare favorably to those of larger systems.

There are two modes: Edit Mode and Command Mode. Edit Mode is always used to input a program for the first time, but either mode can be used to change existing programs after loading them from the disk or typing them in Edit Mode.

Edit Mode is entered directly from the menu. The screen is a 40 × 24 window on the source program. Function keys allow you to move this window to the right or left in 20-character increments, or up and down 24 lines at a time. (Since most of my Assembly Language programs have fewer than 40 characters per line, I tend to view the leftmost 40 characters and make heavy use of the up and down scrolling). The four cursor keys are enabled in Edit Mode, making it especially easy to correct typographical errors. Whole lines can be inserted into the text by moving the cursor to the adjacent line and pressing the Insert function key; a new blank line is inserted, and the user simply types in a new line. Similarly, a whole line can be deleted by moving the cursor there and pressing the Delete function key; the line is removed and the line numbers of the following lines are automatically decremented. There are also keys for inserting or deleting characters. A Tab key is also provided for tabbing to columns 8 and 16. Edit Mode makes it very easy to enter new programs because the user can both type the source program in a natural manner and correct errors and omissions as they occur. Edit Mode is exited via the Back function key, which puts the Editor into Command Mode.

Command Mode reminds me of the UCSD Pascal editor. The first line of the screen shows the Command Mode options: **E**scape, **F**ind, **R**eplace, **M**ove, **I**nsert, **C**opy, **D**elete, **S**how, and **A**djust. Line 2 is reserved for parameters to be input by the user, so in this mode the text window is 40 × 22. Most options require further information to be given on line 2, and very clear prompts given so the user knows what line to enter.

Each option is selected by typing the first character of the option name. For example, to find an occurrence of a string in the source program, the user enters **F**. The system responds with the prompt <count>< (start col, end col)>/string/. To find the second occurrence of the string ABCD between columns 1 and 50, the user would type 2(1,50) /ABCD/. The system would then display the section of the text containing the second such occurrence of ABCD (if any) with the cursor over the A. The symbols < > in the prompting message indicate optional parameters. To find the next occurrence of the string ABCD in the whole source program, the user need only type /ABCD/. The **R**eplace option is like **F**ind, except that each specified occurrence of the string is replaced by a second string given by the user. **R**eplace includes an optional verify operator which allows the user to say yes or no to each replacement. The **M**ove option allows the user to move sections of text, indicated by an interval of line numbers, to a different place in the source program. **C**opy is similar, except that the section of text ends up in both the original position and the new position. **D**elete allows easy removal of several contiguous lines from the text. **I**nsert takes a file from disk and places it anywhere you want in the program being edited. **S**how is a way of moving the window so that a certain line number is at the top of the screen. **A**djust is an easy way to make the line numbers disappear so that the window shows the source program only. **E**scape gets you out of Command Mode and back to the Editor's menu, where you can choose to save the source program to disk, print it, purge it or edit the same or another program.

The Editor performs all line numbering automatically as lines are entered and maintains these numbers in sequence as lines are added or deleted. The user can refer to them for operating on sections of the program; they also appear

```
         Larger system (TXMIRA):
         LI      2,0           MOVE 0 TO REGISTER 2 FOR INDEX
         LI      12,>CO        SET CRU BASE ADDRESS FOR SCREEN
         SBO     >F            SELECT CRU WORD 1
         LDCR    @ZERO,11      MOVE CURSOR TO HOME POSITION
         SBZ     >F            SELECT CRU WORD 0
LOOP     LDCR    @AB(2),7      PUT CHARACTER ON CRU LINE
         SBZ     >8            STROBE CHARACTER TO SCREEN
         SBZ     >A            INCREMENT CURSOR POSITION
         INC     2             ADD 1 TO INDEX REGISTER
         CI      2,2           COMPARE REGISTER 2 TO 2
         JLT     LOOP          LOOP IF MORE CHARACTERS

         . . .
ZERO     DATA    0             DATA DEFINITIONS
AB       TEST    'AB'

         TI-99/4A assembler:
         REF     VMBW .        EXTERNAL REFERENCE TO ROUTINE UTILITY
         . . .
         LI      0,0           VDP RAM ADDRESS = 0 FOR HOME POSITION
         LI,     1,AB          REGISTER 1 POINTS TO FIRST CHARACTER TO DISPLAY
         LI      2,2           REGISTER 2 = NUMBER OF BYTES TO WRITE
         BLWP    @VMBW         CALL UTILITY ROUTINE TO WRITE STRING
         . . .
AB       TEXT    'AB'          DATA DEFINITION
```

Figure 1

on the Assembler output listing, which is handy for debugging.

TI has incorporated most of the features found in editors for larger systems into the 99/4A Editor. In fact, the abilities to edit at the character, line, and group-of-lines levels are not always *all* available in larger editors. The only feature missing from the 99/4A Editor is a variable right margin—a feature which is really not too significant for Assembly Language source programs. [But that would be nice for word processing applications, since this editor already performs 95% of what most people would need for correspondence and document preparation.—Ed.]

The Assembler

The Assembler is a program which converts Assembly Language source programs into object form—the machine-language program that executes on the TI-99/4A. The object program is written to disk. Optionally, a user can print out or write an Assembly Language listing to disk.

The 99/4A Assembler is a lot like the 9900 Assembler, TXMIRA, which runs on larger TI systems. See sample listing in Figure 1. A programmer who is familiar with TXMIRA will be able to write Assembly Language programs for the 99/4A without too much difficulty since the same addressing modes are used and most of the instructions operate in the same way.

One big difference, as might be expected, is in the way a programmer handles input and output to the monitor. The 99/4A Editor/Assembler package includes three groups of built-in subroutines, or *macros*: (1) *Utility Routines* for accessing machine resources, such as screen I/O; (2) *Extended Utilities*, for accessing routines built into the console ROMs and GROMs; and (3) *Basic Support Utilities* for accessing the parameter list in CALL LINK statements from Extended BASIC. These utilities make it unnecessary to use the CRU (Communications Register Unit) lines to the monitor. Under TXMIRA, all peripheral devices are addressed via a fairly complex arrangement of CRU lines. Each device has its own CRU base address and CRU bit assignments, which means that a programmer must have very specific information about each device in order to perform any input or output. On the 99/4A Assembler these difficulties in handling the screen have been eliminated by the Utility Routines. By loading a few registers and invoking the proper utility, a programmer can handle screen I/O in a much simpler way. Figure 1 has the code segments which might be used for writing the character AB to the upper left portion of the screen.

You can see that the Utility Routines really make screen handling easier: You can focus your attention on merely the VDP RAM (the memory associated with the 99/4A monitor) addresses, and not have to worry about the logistics of the move. Furthermore, there is no apparent loss of execution speed in doing it this way.

Another difference between the 99/4A Assembler and those for larger TI computers is that the IDLE instruction is not implemented on the 99/4A. This causes no great difficulty, but it is useful to know. The IDLE instruction just causes the computer to wait for an interrupt; this can be done via another Utility Routine or other means, depending on which device will cause the interrupt.

The optional listing produced by the 99/4A Assembler is quite complete. Statement sequence numbers, source statements, and the hexadecimal code generated are all shown clearly. A symbol table can also be given and, of course, the number of errors is shown. Each error is also flagged in the body of the listing with a descriptive message. One very nice—and all too uncommon—feature is that a display of the number of errors is on the monitor when the Assembler is finished.

Running and Debugging

Once a program has been input, edited, and assembled with no errors, it can be loaded and run by choosing this option from the menu. Another menu option (RUN PROGRAM FILE) allows the user to run programs which were assembled on other Texas Instruments systems or previously assembled on your system.

The Editor/Assembler package has a special debugging utility called DEBUG, which can be very helpful in isolating program errors. For instance, the commands in DEBUG allow you to set *breakpoints* in your program. When the program hits a breakpoint and stops execution, you can then use other commands to examine the contents of memory locations and registers, the Workspace Pointer, the Status Register, or the Program Counter, and if necessary change them to alter the program's execution. DEBUG commands will also allow you to search memory locations for a specific value, or to search memory locations and print those which *don't* have a specific value. DEBUG allows you to begin executing your program at any point you determine; combined with the breakpoints, this allows you to go through a program section by section. All in all, DEBUG provides a good repertoire of useful tools which will make it easier to find out why the program you wrote isn't working the way you thought it would.

PART 2: Fundamentals of Assembly Language Programming on the TI-99/4A

In Part I we gave you a preliminary look at TI's Editor/Assembler for the TI-99/4 and TI-99/4A and mentioned briefly the advantages of programming in Assembly Language. Now let's explore the benefits of Assembly Language more fully by comparing some programs written in Assembly Language and BASIC.

Some Assembly Language Explanations

Before examining some programs, it would be useful to mention some general characteristics of the TMS9900 processor, and then some specifics on the structure of the TI-99/4A.

All 9900 programs make use of 16 workspace registers, each containing 16 bits (one word). Assembly Language programs define 16 contiguous words of memory for these workspace registers and set the hardware register called the Workspace Pointer to point to the first of these memory locations. Having these workspace registers resident *in memory* rather than in the CPU is one of the most powerful features of the 9900-family processors. In an Assembly Language program, the hexadecimal numbers 0 through F refer to the current workspace registers. (In addition, an Assembly Language option allows you to refer to them as R0 through R15, which makes programs easier to read.)

The structure of the memory of the 99/4A is fairly complex. The following explanations cover concepts necessary to understanding the programs in this article, but they only begin to scratch the surface of the memory structure.

CPU RAM (Random Access Memory) resides in the console and is directly addressable by Assembly Language programs. Workspace registers and other memory locations, as well as the programs themselves, reside in CPU RAM.

VDP (Video Display Processor) RAM, also located in the console, takes care of the video screen. Sprites, colors, character patterns, and the screen image itself all reside in VDP RAM. Unlike CPU RAM, however, VDP RAM is not directly addressable by Assembly Language programs. VDP RAM is accessed through specifically assigned CPU RAM addresses. This is called memory mapping. Locations 0 through >02FF in VDP RAM contain the screen image. (The symbol ">" means hexadecimal notation; >02FF=767 in decimal notation.) This means that whatever characters reside in this section of VDP RAM are visible on the screen. To change the screen, the programmer would place the desired character code(s) into VDP RAM at the corresponding location(s). VDP RAM location 0 corresponds to the home position (upper left) on the screen; location 48 (or>30) corresponds to the position called row 2 and column 17 in BASIC. Let's say you want to put an * on the screen at row 2, column 17. The ASCII code for * is 42, or >2A, and the desired VDP RAM location is >30. You might be tempted to use a MOVB (Move Byte) instruction to accomplish this, but remember, the VDP RAM cannot be directly addressed from your Assembly Language program. To access VDP RAM, you'll need to use a Utility Routine. VSBW (VDP Single Byte Write) is a *macro instruction* which places the most significant (leftmost) byte of workspace register 1 at the VDP RAM address contained in register 0. Therefore, to place the * at row 2, column 17, you'd write:

```
REF     VSBW       UTILITY REFERENCE
 .
 .
 .
LI      0,>30      R0=VDP RAM ADDRESS
LI      1,>2A00    RI CONTAINS * IN MSB
BLWP    @VSBW      MOVE TO VDP RAM
```

Most of the utilities use similar schemes of loading data into certain registers and calling the utility by name. I'll talk more about some specific ones later.

The Game of Life

Life is a classic computer game. It is based on the idea of a population which goes through life cycles to form new generations; each position on the screen corresponds to a cell in the population. Cells which are alive are filled in (with asterisks in my example); dead cells are blank. The life cycle, or rules of the game, are applied to each generation to obtain the next generation, and then the new generation is displayed on the screen. The rules of the game determine birth, death, or survival of individual cells, and depend on the state of each cell's 8 neighbors (adjoining cells, considered horizontally, vertically, and diagonally) as follows:

1. A live cell with 2 or 3 neighbors survives to the next generation.
2. A live cell with 0 or 1 neighbor dies of loneliness; a live cell with more than 3 neighbors dies of overcrowding.

The rules are applied to a generation as a whole, before the next generation is displayed. Depending on the initial population, you may see a colony which goes on changing forever, one which dies out or becomes static after a few generations, or one which oscillates among a few patterns.

There are a few restrictions on my implementation of *Life* which should be explained. First, I have defined the initial population in the programs, whereas other versions might allow the user to enter the initial population on the screen at the beginning of the game. In order to be sure the colony does not exceed the size of the 99/4A screen, which is 32 × 24, I have forced the border (rows 1 and 24 and columns 1 and 32) always to remain blank. This means that when the colony becomes large it may lose its symmetry as one side of the colony hits the border.

The two programs which follow are in BASIC (Listing 1) and in Assembly Language (Listing 3). Both follow the same strategy: display the initial colony, calculate the next generation by considering the neighbors of each cell in turn, clear the screen, display the new generation, and loop back to calculate the next generation. The Assembly Language version uses one byte to represent each cell; the BASIC version uses one entry in array SCRN for each cell. At the start of each generation, live cells contain the value 1 and dead cells contain 0. During the calculation of the next generation, a cell can have the values 0 through 3 as follows:

0 = cell is dead and remains dead for the next generation
1 = live cell survives to the next generation
2 = dead cell will be born in the next generation
3 = live cell will die in the next generation

It is necessary to have these four possible values during the calculation so that the program can have the information about the current state of each cell while calculating and storing the next state of each cell. Just before the new generation is displayed (or not displayed if dead), the values of the cells are reset to 0 or 1 by means of the array AFTER.

In examining both versions of *Life* which follow (Listings 1 and 3), you might wonder why anyone would use the more esoteric Assembly Language over the easier-to-understand BASIC. The answer is simple: *speed*. On the 99/4A, the BASIC program takes 2 minutes and 26 seconds between generations; the Assembly Language program takes *less than one second*! The BASIC version is no fun at all to watch, whereas the Assembly Language program provides fine entertainment. [The use of the Utility Routine VMBW (VDP Multiple Byte Write) in the Assembly language is partly responsible for this speed. It shows each new generation all at once. And fortunately, the monitor program is smart enough to capitalize on this by showing only the changed portions of the screen, rather than re-drawing the whole screen each time. If fast enough, the human brain's "persistence of vision" allows us to see individual frames of moving images as continuous rather than discrete pictures—thus making realistic animation sequences truly possible.—Ed.]

Using Assembly Language to Move Sprites

The ability to create sprites which move automatically is one of the best features of the 99/4A. Sprites can be used in Extended BASIC and in Assembly Language programs.

VDP RAM has several areas dedicated to sprites. The Sprite Attribute Block, which gives the sprite locations, sprite numbers, and colors, starts at address >300. Each entry in the Sprite Attribute Block occupies four bytes. A terminator byte with value >0D denotes the end of the Sprite Attribute Block. The Sprite Descriptor Block contains the sprite patterns (shapes), with 8 bytes for each possible sprite. Although the Sprite Descriptor Block starts at VDP RAM address 0 by default, we have already seen that VDP RAM locations 0 through >02FF are used for the screen image table, and locations >0300 through >03FF for the sprite Attribute Block. In order to avoid writing over these areas, the Sprite Descriptor Block usually starts at location >0400 for practical purposes. The entries in the Sprite Descriptor Block are defined to correspond to sprite numbers starting at 0 and occupying 8 bytes each; therefore the entry at location >0400 is for sprite number >80. Thus in Assembly Language programs, the lowest sprite number is usually >80. The Sprite Motion Table, which gives the x- and y-velocities of defined sprites, resides at VDP RAM location >0780. Each entry in the Motion Table occupies four bytes, the last two of which are for system use. The Sprite Motion Table is filled only if automatic motion is to be used. An Assembly Language program could move the sprites (non-automatically) by changing the x- and y-locations of the sprites in the Sprite Attribute Block. But the system is able to move the sprites for you via an interrupt processing routine: Each time a VDP interrupt occurs (60 times per second), the interrupt processing routine moves any eligible sprites according to the Sprite Motion Table. In order to make use of this facility, the Assembly Language program must also load the number of moving sprites at CPU RAM address >837A and enable the VDP interrupts.

Assembly Language vs Extended BASIC

You are probably thinking that this sounds like a lot of work to achieve moving sprites, especially compared to the simple CALL SPRITE statement of Extended BASIC. However, there are times when an Extended BASIC program is inadequate. Coincidence checking in Extended BASIC is not as responsive to velocity changes as you might like.

The programs which follow (Listings 2 and 4) illustrate how Assembly Language can be used to overcome these deficiencies. The program simply moves a target from left to right on the screen while shooting an arrow from the top of the screen to the bottom. Both sprites wrap around the screen. Whenever the arrow hits the target, the sprites stop moving, the target changes to an X, and the program delays long enough to make the blow-up visible. Then the program starts over. The Extended BASIC program relies on CALL COINC to detect hits. You'll notice, however, that the program doesn't seem to detect all hits. The Assembly Language program can stop the action by disabling the VDP interrupt while it checks for coincidence by comparing the locations of the arrow and the target from the Sprite Attribute Block. Moreover, the Assembly Language program can check the point of the arrow against the target instead of checking the upper lefthand corners of the sprites.

Because of these differences, the Assembly Language program appears to detect more hits correctly. Of course, this stop-motion processing must slow down the motion, but it is not noticeable to me. (One indication of the speed of Assembly Language program execution is the large number of statements executed in LOOP2 while the hit shape briefly remains on the screen.)

Another shortcoming of the Extended BASIC version is that the hit shape appears quite a bit to the right of its actual position when the hit occurred. That is because the sprites have continued to move while two BASIC statements (lines 190 and 200) are interpreted and executed. The Assembly Language version has already stopped the motion by disabling the VDP interrupt program via LIMI 0; it doesn't start the motion again until after the hit sequence is complete. Thus, only the Assembly Language program actually shows the blow-up in the right place on the screen.

Understanding An Assembler Listing

The Assembly Language listing (Figure 4) was output by the 99/4A Assembler. You'll notice that the Assembler has added a page number and short title at the top of each page and added a cross-reference list and number-of-errors-found-during-assembly message to the end. The cross-reference list shows the location of the symbols used in the program relative to the beginning of the program. The line numbers in the first column were supplied by the Editor when the program was input and passed along by the Assembler. The second column of the listing shows the relative memory location where each statement or data area will reside during program execution. The third column was also supplied by the Assembler and shows the machine language generated by the Assembly Language statement to the right. The machine language (or *object code*) is expressed in hexadecimal notation with one word per line. The Assembly Language source program (or *source code*) itself starts in the fourth column, which contains the labels. The fifth column contains the source program opcodes, and the sixth column contains the operands. The seventh column contains comments, and other comments are sprinkled throughout the program with asterisks in column 1. *Only the fourth through seventh columns comprise the Assembly Language source program; this is the only part entered by the programmer.* The Assembler generates the rest.

The Utility Routines VMBW, VSBW, VWTR, and VMBR are used in the example program. The VDP Multiple Byte Write (VMBW) moves the number of bytes in register 2 (R2) from the CPU RAM address in R1 to the VDP RAM address in R0. VSBW, the VDP Single Byte Write routine, was explained earlier. VDP Write To Register (VWTR) puts the value that is in the rightmost byte of R1 into the VDP register whose number is in the leftmost byte of R1. Among other things, these VDP registers are used to select VDP modes and features. VMBR is the VDP Multiple Byte Read routine, which reads the number of bytes specified in R2 into the CPU RAM location in R1 from the VDP RAM location in R0.

The logic for detecting hits in the Assembly Language program is based on the fact that the point of the arrow is three pixels to the right and seven pixels below the corner of the sprite which is obtained from the Sprite Attribute Block.

Conclusion

Although they are more complex to write, Assembly Language programs are far superior to BASIC programs when it comes to execution speed and for controlling the facilities of the 99/4A computer. In some cases, as in the game of *Life*, the faster speed of Assembly Language turns a boring game into one which is fun to watch. In other cases, as in the program SHOOT, Assembly Language is capable of providing more accurate results. Thus, having the capability to write programs or subroutines in Assembly Language lets you achieve results which are impossible with BASIC and Extended BASIC alone.

Listing 1 *Life*

```
100 CALL CLEAR
110 DIM OFFSETS(8),AFTER(4)
120 FOR I=1 TO 8
130 READ OFFSETS(I)
140 NEXT I
150 DATA -33,-32,-31,-1
160 DATA 1,31,32,33
170 DIM SCRN(768)
180 REM  INITIALIZE
190 FOR I=1 TO 768
200 SCRN(I)=0
210 NEXT I
220 AFTER(0)=0
230 AFTER(1)=1
240 AFTER(2)=1
250 AFTER(3)=0
260 REM  INITIALIZE POPULATION
270 READ NUMSUB
280 FOR I=1 TO NUMSUB
290 READ ROW,COL
300 ISUB=(ROW-1)*32+COL
310 SCRN(ISUB)=1
320 CALL HCHAR(ROW,COL,42)
330 NEXT I
340 DATA 7
350 DATA 11,16,12,15,12,17,13,14,13,18
    ,14,14,14,18
360 REM  CALCULATE NEXT GENERATION
370 ISUB=34
380 FOR ROW=2 TO 23
390 FOR COL=2 TO 31
400 CNT=0
410 FOR K=1 TO 8
420 M=SCRN(ISUB+OFFSETS(K))
430 IF M=0 THEN 460
440 IF M=2 THEN 460
450 CNT=CNT+1
460 NEXT K
470 IF SCRN(ISUB)=1 THEN 500
480 IF CNT=3 THEN 520
490 GOTO 530
500 IF CNT=2 THEN 530
510 IF CNT=3 THEN 530
520 SCRN(ISUB)=SCRN(ISUB)+2
530 ISUB=ISUB+1
540 NEXT COL
550 ISUB=ISUB+2
560 NEXT ROW
570 REM  SHOW NEW GENERATION
580 CALL CLEAR
590 ISUB=34
600 FOR ROW=2 TO 23
610 FOR COL=2 TO 31
620 SCRN(ISUB)=AFTER(SCRN(ISUB))
630 IF SCRN(ISUB)=0 THEN 650
640 CALL HCHAR(ROW,COL,42)
650 ISUB=ISUB+1
660 NEXT COL
670 ISUB=ISUB+2
680 NEXT ROW
690 GOTO 370
700 END
```

Listing 2 *Shoot an Arrow*

```
100 CALL CLEAR
110 REM  DEFINE SPRITES
120 CALL CHAR(142,"FF81BDA5A5BD81FF")
130 CALL CHAR(143,"181818181818183C18")
140 CALL CHAR(141,"8142241818244218")
150 CALL SPRITE(#1,142,7,124,1,0,100)
160 CALL SPRITE(#2,143,2,1,124,127,0)
170 REM  TEST FOR HIT
180 CALL COINC(#1,#2,10,HIT)
190 IF HIT=0 THEN 180
200 CALL MOTION(#1,0,0)
210 CALL MOTION(#2,0,0)
220 CALL PATTERN(#1,141)
230 FOR DELAY=1 TO 50
240 NEXT DELAY
250 GOTO 150
260 END
```

Listing 3 *Life*

```
        IDT 'LIFEA'
        DEF LIFEA
        REF VMBW
WS      BSS 32
SCRN    BSS 768
GENSCR  BSS 768
OFSET   DATA -33,-32,-31,-1
        DATA 1,31,32,33
FSTGEN  DATA 7,335,366,368,397,401,429,433
H00     BYTE >00
H01     BYTE >01
H02     BYTE >02
BLNK    BYTE >20
STAR    BYTE >2A
AFTER   BYTE 0,1,1,0
        EVEN
H2000   DATA >2000
LIFEA   LWPI WS                 START OF PROGRAM
*CLEAR  SCREEN ARRAY.
        LI R1,766               LOOP COUNTER AND INDEX
CLEAR   CLR @SCRN(R1)           CLEAR WORD
        DECT R1                 POINT TO WORD
        JLT  INIT               DONE
        JMP CLEAR
```

Listing 3 *Life* continued

```
*LOAD INITIAL GENERATION AND DISPLAY.
INIT   MOV  @FSTGEN,R3   R3=#OF CELLS
       A    R3,R3        DOUBLE IT FOR WORDS
INITLP MOV @FSTGEN(R3),R4  R4 CONTAINS OFFSET
       MOVB @H01,@SCRN(R4)    SCREEN POSITION =1
       DECT R3
       JNE INITLP        MORE TO DO
       BL   @SHOWIT      SHOW INITIAL GEN
       LIMI 2            ENABLE VDP INTERRUPT FOR QUIT
*CALCULATE NEXT GENERATION.
CLCGEN LI   R1,33        INDEX(ISUB)
       LI   R3,22        OUTER LOOP CTR(ROW)
CLCLP  LI   R4,30
*COUNT NEIGHBORS.
CLCNBR LI   R5,0         NEIGHBORS COUNTER(CNT)
       LI   R6,0         LOOP CONTROL,INDEX TO OFSET
NBRS   MOV  R1,R7        COPY TO WORK ON
       A    @OFSET(R6),R7     R7->DISP OF NEIGHBOR
       CB   @SCRN(R7),@H00    NBR=0?
       JEQ  NXTNBR            YES
       CB   @SCRN(R7),@H02    NBR=2?
       JEQ  NXTNBR            YES
       INC  R5                NEIGHBOR ON
NXTNBR INCT R6
       CI   R6,16             DONE?
       JLT  NBRS              LOOK AT NEXT NEIGHBOR
       CB   @SCRN(R1),@H01    IS CELL ON NOW?
       JEQ  CELLON            YES
       CI   R5,3              3 NEIGHBORS?
       JEQ  CHANGE            YES-BIRTH
       JMP  NOCHG             NO
CELLON CI   R5,2              2 NEIGHBORS?
       JEQ  NOCHG             YES-SURVIVE
       CI   R5,3              3 NEIGHBORS?
       JEQ  NOCHG             YES-SURVIVE
CHANGE AB   @H02,@SCRN(R1)    BIRTH OR DEATH
NOCHG  INC  R1                NEXT CELL
       DEC  R4                NEXT COL
       JNE  CLCNBR
       INCT R1                SKIP TWO EDGE CELLS
       DEC  R3                NEXT ROW
       JNE  CLCLP
*RESET SCRN ELEMENTS TO 0 FOR DEAD, 1 FOR ALIVE.
       LI R5,33               INDEX TO SCRN(ISUB)
       LI R3,22               ROW CTR
LOOP   LI R4,30
LOOP1  MOVB @SCRN(R5),R6       R6=CELL VALUE IN MSB
       SRL  R6,8              SHIFT TO LSB
       MOVB @AFTER(R6),@SCRN(R5)   CHANGE CELL TO 0 OR 1
       INC  R5                NEXT CELL
       DEC  R4                NEXT COL
       JNE  LOOP1
       INCT R5
       DEC  R3
       JNE  LOOP
       BL   @SHOWIT           SHOW NEW GENERATION
       JMP  CLCGEN            CALC NEXT GEN
*SUBROUTINE TO DISPLAY GENERATION ON SCREEN.
SHOWIT LI   R5,767            R5 INDEXES BOTH SCRN
*                               &GENSCR.
BLDSCR CB   @H00,@SCRN(R5)    IS BYTE 0 (DEAD)?
       JEQ  BLK               YES
       MOVB @STAR,@GENSCR(R5) NO-PUT * IN GENSCR
       JMP  NXTPOS
BLK    MOVB @BLNK,@GENSCR(R5) PUT BLANK IN GENSCR
NXTPOS DEC  R5                POINT TO NEXT CELL
       JLT  OUTSCR            DISPLAY IF DONE
       JMP  BLDSCR            LOOP IF NOT DONE
OUTSCR CLR  R0                VDP RAM ADDRESS (HOME)
       LI   R1,GENSCR         GENSCR CONTAINS DISP DATA
       LI   R2,768            768 BYTES TO WRITE
       LIMI 0
       BLWP @VMBW             WRITE SCREEN
       LIMI 2
       B    *R11               RETURN
       END  LIFEA
```

Listing 4 *Shoot an Arrow*

```
 99/4 ASSEMBLER
VERSION 1.2                                                    PAGE 0001
 0001                       IDT   'SHOOTA'
 0002                       DEF   SHOOTA
 0003                       REF   VMBW,VSBW,VWTR,VMBR
 0004 0000        WS        BSS   32
 0005 0020   7C   SAL       BYTE  >7C,>01,>80,>06    SPRITE 1 LOCN AND COLOR
      0021   01
      0022   80
      0023   06
 0006 0024   01             BYTE  >01,>7C,>81,>01    SPRITE 2 LOCN AND COLOR
      0025   7C
      0026   81
      0027   01
 0007 0028   D0             BYTE  >D0                TERMINATOR
 0008 0029   FF   SHAPE     BYTE  >FF,>81,>BD,>A5,>A5,>BD,>81,>FF   TARGET
      002A   81
      002B   BD
      002C   A5
      002D   A5
      002E   BD
      002F   81
      0030   FF
 0009 0031   18             BYTE  >18,>18,>18,>18,>18,>18,>3C,>18   ARROW
      0032   18
      0033   18
      0034   18
      0035   18
      0036   18
      0037   3C
      0038   18
 0010 0039   81   HITSHP    BYTE  >81,>42,>24,>18,>18,>24,>42,>81   HIT SHAPE
      003A   42
      003B   24
      003C   18
      003D   18
      003E   24
      003F   42
      0040   81
 0011 0041   00   SPEED     BYTE  >00,>64,>00,>00    SPRITE 1 VELOSITY
      0042   64
      0043   00
      0044   00
 0012 0045   7F             BYTE  >7F,>00,>00,>00    SPRITE 2 VELOSITY
      0046   00
      0047   00
      0048   00
 0013 0049   00   H00       BYTE  >00
 0014 004A   02   H02       BYTE  >02
 0015 004B        Y1        BSS   1
 0016 004C        X1        BSS   1
 0017 004D        DUMMY     BSS   2
 0018 004F        Y2        BSS   1
 0019 0050        X2        BSS   1
 0020 0051   03   H03       BYTE  >03
 0021 0052   07   H07       BYTE  >07
 0022                       EVEN
 0023 0054 0020   H0020     DATA  >0020
 0024 0056 02E0   SHOOTA    LWPI  WS
      0058 0000'
 0025             *FILL SCREEN WITH BLANKS.
```

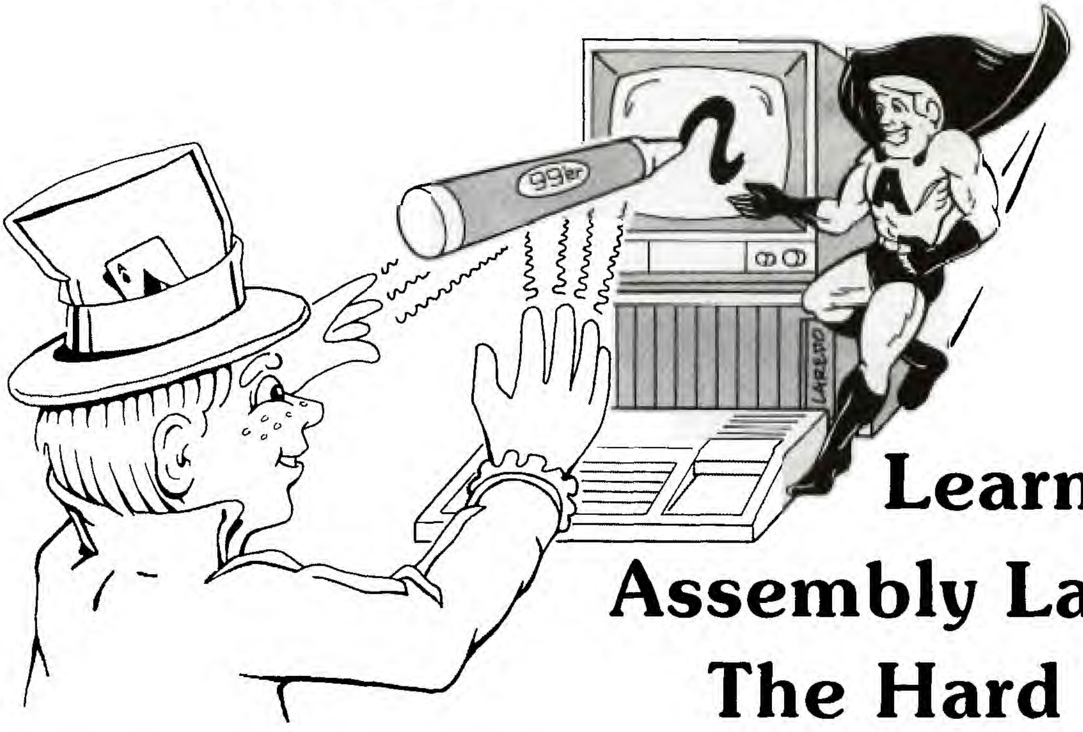
Listing 4 *Shoot an Arrow* continued

```
 99/4 ASSEMBLER
VERSION 1.2                                                      PAGE 0002
  0026 005A 04C0         CLR  0               VDP RAM SCREEN HOME
  0027 005C 0201         LI   1,>2000         BLANK IN MSB OF R1
       005E 2000
  0028 0060 0420  BLNKIT BLWP @VSBW           WRITE BLANK
       0062 0000
  0029 0064 0580         INC  0
  0030 0066 0280         CI   0,768           DONE?
       0068 0300
  0031 006A 11FA         JLT  BLNKIT          NOT YET
  0032            *SET UP VDP REGISTER 1
  0033 006C 0200         LI   0,>01E0         NORMAL SIZED SPRITES
       006E 01E0
  0034 0070 0420         BLWP @VWTR
       0072 0000
  0035            *SET UP SPRITE ATTIBUTE BLOCK.
  0036 0074 0201  DEFSPR LI   1,SAL           R1-MY ATTRIBUTE LIST
       0076 0020'
  0037 0078 0200         LI   0,>0300         R0->ADDRESS OF VDP SAB
       007A 0300
  0038 007C 0202         LI   2,9             9 BYTES TO WRITE
       007E 0009
  0039 0080 0420         BLWP @VMBW           WRITE TO VDP RAM
       0082 0000
  0040            *LOAD SPRITE DEFINITIONS
  0041 0084 0201         LI   1,SHAPE         R1->MY SPRITE SHAPES
       0086 0029'
  0042 0088 0200         LI   0,>0400         ADDRESS OF FIRST SPRITE
       008A 0400
  0043 008C 0202         LI   2,16            16 BYTES TO MOVE
       008E 0010
  0044 0090 0420         BLWP @VMBW           WRITE TO VDP RAM
       0092 0082'
  0045            *SET UP SPRITE MOTION TABLE.
  0046 0094 0200         LI   0,>0780         R0->MOTION TABLE IN VDP RAM
       0096 0780
  0047 0098 0201         LI   1,SPEED         R1->MY SPEED DATA
       009A 0041'
  0048 009C 0202         LI   2,8             8 BYTES TO MOVE
       009E 0008
  0049 00A0 0420         BLWP @VMBW           WRITE
       00A2 0092'
  0050            *SET NUMBER OF MOVING SPRITES.
  0051 00A4 D820         MOVB @H02,@>837A     2 MOVING SPRITES
       00A6 004A'
       00A8 837A
  0052            *MAKE SPRITES MOVE BY INTERRUPT FROM 9901 I/O BOARD.
  0053 00AA 0300  MOVEIT LIMI 2               ENABLE INTERRUPT
       00AC 0002
  0054            *CHECK FOR COINCIDENCE.
  0055 00AE 0300         LIMI 0               DISABLE VDP INTERRUPT
       00B0 0000
  0056            *GET SPRITE POSITIONS.
  0057 00B2 0200         LI   0,>0300         R0->Y OF SPRITE IN VDP RAM
       00B4 0300
  0058 00B6 0201         LI   1,Y1            BUFFER FOR READ
       00B8 004B'
  0059 00BA 0202         LI   2,6             6 BYTES TO READ
       00BC 0006
  0060 00BE 0420         BLWP @VMBR           READ FROM VDP RAM
```

Listing 4 *Shoot an Arrow* continued

```
 99/4 ASSEMBLER
VERSION 1.2                                                          PAGE 0003
         00C0 0000
  0061                *CHECK COLUMNS FOR X1<=X2+3<=X1+7
  0062   00C2 B820           AB   @H03,@X2             X2=X2+3
         00C4 0051'
         00C6 0050'
  0063   00C8 7820           SB   @X1,@X2              X2+X2-X1
         00CA 004C'
         00CC 0050'
  0064   00CE 11ED           JLT  MOVEIT               NO HIT IF RESULT >0
  0065   00D0 9820           CB   @X2,@H07             COMPARE TO 7
         00D2 0050'
         00D4 0052'
  0066   00D6 15E9           JGT  MOVEIT               NO HIT IF RESULT >7
  0067                *CHECKS ROWS FOR Y1<=Y2+7<=Y1+7.
  0068   00D8 B820           AB   @H07,@Y2             Y2=Y2+7
         00DA 0052'
         00DC 004F'
  0069   00DE 7820           SB   @Y1,@Y2              Y2=Y2-1
         00E0 004B'
         00E2 004F'
  0070   00E4 11E2           JLT  MOVEIT               NO HIT IF RESULT <0
  0071   00E6 9820           CB   @Y2,@H07
         00E8 004F'
         00EA 0052'
  0072   00EC 15DE           JGT  MOVEIT               NO HIT IF RESULT >7
  0073                *HIT
  0074                *CHANGE SPRITE DEFINITIONS.
  0075   00EE 0201           LI   1,HITSHP             R1->HIT SHAPE
         00F0 0039'
  0076   00F2 0200           LI   0,>400               R0->VDP RAM
         00F4 0400
  0077   00F6 0202           LI   2,8                  8 BYTES TO LOAD
         00F8 0008
  0078   00FA 0420           BLWP @VMBW                WRITE TO VDP RAM
         00FC 00A2'
  0079                *WAIT TO LET BLOW UP BE SEEN.
  0080   00FE 0203           LI   3,10                 OUTER LOOP CTR
         0100 000A
  0081   0102 0202    LOOP2A LI   2,12000              LOOP CUONTER
         0104 2EE0
  0082   0106 0602    LOOP2  DEC  2                    DECREMENT
  0083   0108 16FE           JNE  LOOP2                WAIT MORE
  0084   010A 0603           DEC  3                    DECREMENT OUTER CTR
  0085   010C 16FA           JNE  LOOP2A               WAIT MORE
  0086   010E 10B2           JMP  DEFSPR               START OVER
  0087                       END  SHOOTA
L
 99/4 ASSEMBLER
VERSION 1.2                                                        PAGE 0004
   ' BLNKIT  0060     ' DEFSPR  0074     ' DUMMY   004D     ' H00      0049
   ' H0020   0054     ' H02     004A     ' H03     0051     ' H07      0052
   ' HITSHP  0039     ' LOOP2   0106     ' LOOP2A  0102     ' MOVEIT   00AA
     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
   ' SAL     0020     ' SHAPE   0029     D SHOOTA  0056     ' SPEED    0041
   E VMBR    00C0     E VMBW    00FC     E VSBW    0062     E VWTR     0072
   ' WS      0000     ' X1      004C     ' X2      0050     ' Y1       004B
   ' Y2      004F
  0000 ERRORS
```

MAGIC CRAYON



Learning Assembly Language The Hard Way

Like many other 99'ers, I was anxious to receive the long-awaited Editor/Assembler package. I remember the excitement of unwrapping the 470 page manual when it arrived—and the sinking feeling when I read, "This manual assumes that you already know a programming language, preferably an assembly language."

My anxiety grew as I thumbed through it—there were no pictures, cartoons, or fill-in-the-blank examples. It did say, "There are many fine books available which teach the basics of assembly language." So I called the local computer stores. The only books they were aware of, however, also assumed familiarity with basics.

I guess I had some fuzzy ideas about assembly language in the back of my mind: It was qualitatively different from higher level languages, requiring an in-depth knowledge of digital electronics and a capacity for the most detailed sort of logical-mathematical thought. In short—nothing seemed more difficult. . .

And my experience thus far seemed to confirm my worst fear. Learning assembly language presumed a prior knowledge of assembly language; it was not merely difficult—it was *impossible.* After running *Tombstone City* a few times and typing in Pat Swift's *Life* program (See "Fundamentals of Assembly Language Programming, Part 1"), I put the Editor/Assembler on a shelf thinking maybe I'd learn about it gradually over the next year or two.

It would still be there gathering dust were it not for a back injury that kept me flat on the floor, unable to do anything *except* read the manual. I was surprised to discover that writing an assembly language program is similar to, and in some respects simpler than, writing a program in BASIC. A new programming context or conceptual model is required. But to get started, I found that this picture could be primitive, containing many over-simplifications and approximations.

The picture I developed enabled me to successfully formulate and execute a simple programming objective. The program and associated underlying concepts are presented here to facilitate the learning process for others who, like me, find it hard to overcome preconceived notions about how difficult assembly language is. The program should not be taken as a model of exemplary programming technique; at this point my conception of "good programming" is programming that works . . . period. You will undoubtedly be able to find ways to improve this one—to make it work faster and utilize memory more efficiently—and in so doing, further develop the concepts presented.

In TMS9900 Assembly Language, four video display modes are available: Graphics (or Pattern) Mode, Text Mode, Bit-Map Mode (99/4A only), and Multicolor Mode. In Multicolor Mode, the screen is divided into a grid 64 × 48, with each box measuring 4 pixels on a side. Each box can have a color assigned to it.

The program allows use of a joystick to move a flashing cursor on the screen. Whenever the fire button is depressed, the cursor leaves a trail of small, colored boxes. The following single key commands are available:

C—*Change Color*. Displays a color palette and pointer. Move the pointer to the desired color with the joystick. Press the fire button to make that the color of the boxes, or press the C key to make it the color of the screen background.

S—*Save Screen*. Saves the current contents of the screen as DSK1.SCREEN.

R—*Recall Screen*. Loads the contents of DSK1.SCREEN for subsequent modification.

E—*Erase Screen*. Erases the screen contents.

T—*Terminate*. Returns to the Master Title Screen.

In order to understand how the program works, it will be helpful to differentiate two systems. You probably know that the Central Processing Unit (CPU) in the Home Computer is the TMS9900. It has three built-in 16-bit "hardware" registers (the Program Counter, Workspace Pointer, and Status Register) and makes use of sixteen workspace registers located in read-write memory. Because these 16-bit workspace registers are not located on the chip, they are called "software" registers. The CPU can directly address the read-write memory (RAM) in the Memory Expansion Unit and CPU scratch pad, as well as ROM in the console, Command Cartridges, and various peripherals. However, it cannot directly address the 16K of RAM built into the console.

The 16K RAM block is addressed by another microprocessor—The TMS9918 (or 9918A if you have a 99/4A). This Video Display Procesor (VDP) has eight 8-bit hardware registers and four 8-bit software registers. The software registers are located in read-write memory locations which can also be addressed by the CPU. The fact that these four bytes can be addressed by both the CPU and VDP makes it possible for the CPU and VDP systems to transfer data back and forth. The CPU addresses of the registers—8800, 8802, 8C00, 8C02—are assigned respectively to the symbols VDPRD (VDP Read Data Address), VDPSTA (VDP Read Status Register), VDPWD (VDP Write Data Address), VDPWA (VDP Write Address).

We don't have to be concerned with the details of moving data to and from VDP RAM and to VDP registers, however, thanks to some of the built-in programs called *utilities*. The five utilities of use are identified by the symbols VSBW, VMBW, VSBR, VMBR, and VWTR. The respective functions of these programs are VDP RAM: Single Byte Write, Multiple Byte Write, Single Byte Read, Multiple Byte Read, and Write to Register. User workspace registers are used to pass parameters—e.g., the number of bytes to read or write—to the utility.

The standard utilization of VDP RAM in the Editor/Assembler is shown on Table 1. The blocks involved in the multicolor mode are the Screen Image and Pattern Descriptor Tables. Before entering multicolor mode, the Screen Image Table is initialized. The 768 bytes of the table are divided into six 128-byte sets. Each set is further subdivided into four 32-byte groups. To initialize the table, the numbers 1-31 are written in order into each of the four 32-byte groups in the first set: 0, 1, 2,. . . 31 four times. Then the numbers 32-61 are written four times into the next 128-byte set. This process is continued until the numbers 160-191 are written four times in the sixth 128-byte set. In my program, I didn't want this process to be visible on the screen, so I first put the display in Text Mode and made the foreground and background colors gray.

Once the Screen Image Table is initialized, color boxes are placed on the screen by means of the Pattern Descriptor Table. Each 4×4 pixel box on the screen corresponds to half a byte in the Pattern Descriptor Table. To place a colored box on the screen, the appropriate color code is written in the nybble (4 bits) in the Pattern Descriptor Table which corresponds to the desired screen position.

Table 1 VDP RAM MEMORY —Editor/Assembler—

Address of First Byte Decimal	Hex	Length of Block, Bytes	Contents
0	>0000	768	Screen Image Table
768	>0300	128	Sprite Attribute List
896	>0380	128	Color Table
1024	>0400	896	Sprite Descriptor Table
1920	>0780	128	Sprite Motion Table
2048	>0800	2048	Pattern Descriptor Table and Peripheral Access Blocks
4096	>1000	10199	More Peripheral Access Blocks and Buffers
14295	>37D7	2089	Reserved for Diskette Device Service Routines
16383	>3FFF	___	Last Address
Total 16384 Bytes			

The first eight bytes of the Pattern Descriptor Table correspond to the boxes in a column beginning in the upper left corner of the screen. The first four bits in byte #1 contain the color of the box in the extreme upper left corner, and the last four bits the color of the box immediately to the right of the first box. Byte #2 contains the colors of the two boxes immediately under the first two, and so on for the first eight bytes.

The ninth byte in the table contains the colors for the pair of boxes in a new column beginning again at the top of the screen. Subsequent bytes follow this pattern corresponding to 32 columns of box pairs with eight pairs in each column. This group of 256 bytes thus takes care of the top sixth of the screen.

The 257th byte corresponds to the beginning of a new column of box pairs starting again on the left side of the screen. The six 256-byte groups thus correspond to the 3,072 possible boxes in multicolor mode. [Since the color of each box is indicated in a name table in memory, and the names are mapped onto the screen according to their position in the table, this multicolor mode is a *true* memory-mapped configuration. It does, however, trade off lower resolution for color memory-mapping capability, but the high-resolution sprites are still available. For an explanation of sprites and an introduction to the high-resolution bit-map mode, see "3-D Animation".—Ed.]

In the program, a double-size sprite provides a reference point for determining where boxes will appear. The dot row and dot column of the sprite can be determined at any time by referring to the Sprite Attribute List in VDP RAM. Then, since boxes are supposed to appear in the center of the sprite, the screen location can be calculated by adding 8 to the dot row and dot column, which represent the sprite's upper left corner. But in order to find the corresponding location in the Pattern Descriptor Table, a few more calculations must be performed.

If we let R and C be the dot row and dot column desired for the box location, the number of complete 256-byte groups above that location is the integer quotient of R/32. Multiplying that number by 256 thus gives the first component of the offset in the Pattern Descriptor Table.

Similarly, the integer quotient of C/8 gives the number of complete 8-byte columns to the left of the location. So that number is multiplied by 8 and added to the offset. Dividing the remainder of R/32 by 4 gives the number of bytes above the location in the 8-byte column the location is in. Adding that to the offset gives the offset for the byte in the Pattern Descriptor Table.

But we still have to know if the desired location is the most or least significant nybble of the byte, and to determine that we can divide the remainder of C/8 by 4. If the integer quotient is 0, it's the left nybble; if 1, it's the right nybble. The appropriate color code then need only be placed in the correct nybble (leaving the other one unchanged), and the box appears just where it should.

Let's consider an example: Suppose the upper left corner of the sprite were at dot row 83 and dot column 147. The center of the sprite would then be at 91 and 155. The number of complete groups (32 columns with 8 bytes in each) above that location is 2, i.e., INT(91/32). So the initial component of the offset is 2 * 256 or 512 bytes. The number of 8-byte columns to the left of the location is INT(155/8) or 19. That makes the offset 531. Above the location, in its 8-byte column, there are 6 bytes—i.e., INT((remainder 91/32)/4)—giving an offset of 537. The remainder of 155/8 is 3, and INT(3/4) is 0, so the nybble of interest is the most significant (left) one of the 539th byte of the Pattern Descriptor Table.

Now let's take a brief look at the source listing. The first section consists of a number of assembler directives. The DEF directive makes the symbol MARKER available to other programs, and the REF directives make several utilities available for use of MARKER. Then there is a variety of other assembler directives. The simplest type is EQUate, which assigns a constant to a symbol at assembly time. USRWS, for >20BA (8378), and that value replaces the symbol wherever it appears in an operand; the label may subsequently be substituted for the number.

The mnemonic BSS stands for Block Starting with Symbol. This directive causes the assembler to advance its location counter without writing anything into the object program. It leaves an empty area (of the number of bytes specified in the operand) which can then be used as a storage space for data later on. The label is set equal to the memory location of the first byte in the block at the time the object program is loaded. (Since this program is relocatable, the place where the loader program decides to start loading it may change, depending on what other programs have already been loaded.)

The DATA, BYTE, and TEXT directives are similar to BSS except that the contents of the buffer are explicitly defined in the operand field. The label is assigned the address of the first byte at the time the object program is loaded. All of these buffer areas are contiguous. For example, look at the instructions immediately after the label MARKER. The pattern codes for two double-size sprites, the cursor and arrow, are loaded into the Sprite Descriptor Table in VDP RAM. Since the pattern data for ARROW is contiguous with that of CURSOR in both CPU and VDP RAM, all 64 bytes can be loaded in one shot.

You should have little trouble figuring out the rest of the program by reading the comments provided and referring to the manual. But don't stop after you understand how it works—try to make some changes. To start with, try changing the shape and colors of the sprite cursor, the arrangement of the color palette on the screen, etc. Then try to make the program more efficient in speed and utilization of memory.

Be prepared to run into problems; it's through encountering and solving them that you'll learn most rapidly. When I decided to stop reading and start trying to write a program, I had visions of seeing a curl of white smoke rise from the computer's cooling vents, but that didn't happen to me and probably won't happen to you either. So don't be afraid to experiment.

Listing 1 *Magic Crayon*

```
        DEF   MARKER
        REF   VSBW,VMBW,VMBR,VSBR
        REF   VWTR,KSCAN,DSRLNK
*
* DEFINITION OF LABELS
*
SCREEN  BSS   >300
PALET   BSS   >600
PATRN   BSS   >600
ROW     BSS   1
COL     BSS   1
CURSOR  DATA  >8040,>2010,>0804,>0000
        DATA  >0000,>0408,>1020,>4080    CH 128
        DATA  >0102,>0408,>1020,>0000
        DATA  >0000,>2010,>0804,>0201
ARROW   DATA  >0102,>0408,>0000,>0000
        DATA  >0000,>0000,>0000,>0000    CH 132
        DATA  >0080,>4020,>0000,>0000
        DATA  >0000,>0000,>0000,>0000
ATTRIB  DATA  >5878,>800F,>D000
ARRATT  DATA  >6578,>8401
PDATA   DATA  >0600,>1000,>0000,>0600
        DATA  >000B
        TEXT  'DSK1.SCREEN'
ZERO    DATA  >0000
D32     DATA  >0020
D8      DATA  >0008
GRAY    DATA  >EEEE
MAX     DATA  >05FF
COLMAX  DATA  >0100
LOAD    BYTE  >05
BLACK   BYTE  >11
ONE     BYTE  >01
TWO     BYTE  >02
FCOLOR  BYTE  >10
BCOLOR  BYTE  >0E
H18     BYTE  >12
H14     BYTE  >0E
H11     BYTE  >0B
H07     BYTE  >07
H06     BYTE  >06
H05     BYTE  >05
H02     BYTE  >02
NOKEY   BYTE  >FF
PAB     EQU   >0F80
USRWS   EQU   >20BA
PNTR    EQU   >8356
UNIT    EQU   >8374
FIRE    EQU   >8375
JOYSTY  EQU   >8376
JOYSTX  EQU   >8377
SPRITE  EQU   >837A
STATUS  EQU   >837C
GPLWS   EQU   >83E0
*
* DEFINE SPRITE PATTERNS FOR CHRS 128 AND 132
*
MARKER  LWPI  USRWS                 LOAD WORKSPACE POINTER / START
        LI    R0,>400               VDP ADDRESS CH 128 SPRITE DESCRIPTOR TABLE
        LI    R1,CURSOR             CPU ADDRESS OF CHAR PATTERN
        LI    R2,64                 64 BYTES TO MOVE (2 PATERNS)
        BLWP  @VMBW                 LOAD DATA TO VDP RAM
*
* SET FOREGROUND AND BACKGROUND TO GRAY
*
        LI    R0,>01F0              PLACE IN TEXT MODE
        BLWP  @VWTR                 WRITE TO VDP R1
        LI    R0,>07EE              SET FORE AND BACKGROUND TO GRAY
        BLWP  @VWTR                 WRITE TO VDP R7
*
* INITIALIZE SCREEN IMAGE TABLE FOR MULTICOLOR MODE
*
        LI    R0,SCREEN             INITIALIZE POINTER
        LI    R1,6                  INITIALIZE GROUP COUNTER
        CLR   R2                    INITIALIZE VALUE
LOOP0   LI    R3,4                  INITIALIZE REPETITIONS COUNTER
LOOP1   LI    R4,>20                INITIALIZE VALUE COUNTER
        MOVB  R2,R5                 START REPETITION
LOOP2   MOVB  R5,*R0+               STORE VALUE IN ARRAY SCREEN
        AI    R5,>0100              CHANGE TO NEXT VALUE
        DEC   R4                    COUNT DOWN FOR NEXT VALUE
        JNE   LOOP2                 DO NEXT VALUE
        DEC   R3                    DEC REPETITION COUNTER
```

Listing 1 *Magic Crayon* continued

```
         JNE   LOOP1                 DO NEXT REPETITION
         AI    R2,>2000              NEXT STARTING VALUE
         DEC   R1                    DEC GROUP COUNTER
         JNE   LOOP0                 DO NEXT GROUP
         LI    R0,>00                VDP ADDRESS FOR SCREEN IMAGE
         LI    R1,SCREEN             CPU ADDRESS OF DATA BUFFER
         LI    R2,>300               768 BYTES TO WRITE
         BLWP  @VMBW                 INITIALIZE VDP SCREEN IMAGE
*
* INITIALIZE COLOR PALETTE SCREEN
*
         LI    R0,>100               INITIALIZE WORD COUNTER
         LI    R1,PALET              INITIALIZE POINTER FOR PALET ARRAY
LOOP3    MOV   @GRAY,*R1+            STORE GRAY COLOR >EEEE
         DEC   R0                    DEC WORD COUNTER
         JNE   LOOP3                 WRITE NEXT WORD
         CLR   R0                    INITIALIZE COLOR VALUE
         LI    R3,16                 INITIALIZE COLOR COUNTER
LOOP4    LI    R4,2                  INITIALIZE COLUMN COUNTER
LOOP5    MOVB  @GRAY,*R1+            STORE GRAY BYTE
         MOVB  @GRAY,*R1+            STORE ANOTHER GRAY BYTE
         MOVB  @BLACK,*R1+           STORE BLACK BYTE
         LI    R5,4                  LOAD COUNTER FOR COLOR BYTES
LOOP6    MOVB  R0,*R1+               STORE A COLOR BYTE
         DEC   R5                    DEC COLOR BYTE COUNTER
         JNE   LOOP6                 STORE ANOTHER COLOR BYTE
         MOVB  @BLACK,*R1+           STORE A BLACK BYTE
         DEC   R4                    DEC COLUMN COUNTER
         JNE   LOOP5                 DO SECOND COLUMN
         SWPB  R0                    SHIFT TO LEAST SIG BYTE
         AI    R0,>11                ADD 1 FOR NEXT COLOR NUMBER
         SWPB  R0                    SHIFT BACK TO MOST SIG BYTE
         DEC   R3                    COUNT DOWN COLOR COUNTER
         JNE   LOOP4                 DO NEXT TWO COLUMNS
         LI    R0,>300               SET BYTE COUNTER FOR REMAINING SCREEN
LOOP7    MOVB  @GRAY,*R1+            STORE A GRAY BYTE
         DEC   R0                    COUNT DOWN
         JNE   LOOP7                 REPEAT UNTIL DONE
*
* INITIALIZE PATTERN TABLE - TRANSPARENT
*
CLEAR    LI    R0,>300               INITIALIZE WORD COUNTER
         LI    R1,PATRN              INITIALIZE POINTER FOR PATTERN ARRAY
LOOP8    MOV   @ZERO,*R1+            STORE COLOR = TRANSPARENT
         DEC   R0                    COUNT DOWN FOR NEXT WORD
         JNE   LOOP8                 WRITE NEXT WORD IN ARRAY
*
* LOAD PATTERN TABLE
*
         LI    R0,>800               VDP PATTERN TABLE ADDRESS
         LI    R1,PATRN              CPU BUFFER ADDRESS
         LI    R2,>600               1536 BYTES TO WRITE
         BLWP  @VMBW                 WRITE TO VDP RAM
*
* SELECT DOUBLE SIZE AND MULTICOLOR MODE
*
         LI    R0,>01EA              TO WRITE 11101010 TO VDP R1
         BLWP  @VWTR                 WRITE TO VDP R1
         SWPB  R0                    MOVE >EA TO MOST SIG BYTE
         MOVB  R0,@>83D4             STORE COPY (>EA) IN CPU RAM
*
* DEFINE ATTRIBUTES FOR SPRITE #0
*
         LI    R0,>300               VDP SPRITE ATTRIBUTE LIST
         LI    R1,ATTRIB             LOCATION OF ATTRIBUTE LIST FOR SPRITE 0
         LI    R2,6                  6 BYTES TO MOVE
         BLWP  @VMBW                 WRITE DATA TO VDP RAM
*
* DEFINE # OF ACTIVE SPRITES
*
         MOVB  @ONE,@SPRITE          STORE NO. OF ACTIVE SPRITES IN CPU RAM
*
* INITIALIZE CURSOR COLOR AND COLOR CHANGE COUNTER
*
         LI    R3,>0F01              SPRITE COLORS - WHITE/BLACK IN R3
         CLR   R4                    INITIALIZE COUNTER - COLOR CHANGE
*
* ---------- START MAIN LOOP ----------------
*
* CHECK JOYST FOR MOTION, FIRE BUTTON AND KEYS
*
CHECK    LIMI  2                     ENABLE INTERRUPTS
         LIMI  0                     DISABLE INTERRUPTS
```

Listing 1 *Magic Crayon* continued

```
        LI   R0,1             INDICATE REPETIONS OF CHECKS
        BL   @CHECKS          BRANCH TO SUBROUTINE CHECKS
        MOVB @ONE,@UNIT       SELECT REMOTE UNIT TO SCAN
        BLWP @KSCAN           SCAN LEFT KEYBOARD
        CB   @FIRE,@H05       WAS "E" PRESSED?
        JEQ  CLEAR            IF YES GO TO CLEAR SCREEN
        CB   @FIRE,@H02       WAS "S" PRESSED?
        JNE  NEXT1            IF NOT, GO ON
        B    @SAVE            IF SO, BRANCH TO SAVE ROUTINE
NEXT1   CB   @FIRE,@H06       WAS "R" PRESSED?
        JNE  NEXT2            IF NOT, GO ON
        B    @RECALL          IF SO, BRANCH TO RECALL ROUTINE
NEXT2   CB   @FIRE,@H11       WAS "T" PRESSED?
        JNE  NEXT3            IF NOT, GO ON
        LIMI 2                ENABLE INTERRUPTS
        LWPI GPLWS            LOAD GPL WORK SPACE
        BLWP @0000            RETURN TO MASTER TITLE SCREEN
NEXT3   CB   @FIRE,@H14       WAS "C" PRESSED?
        JNE  NEXT4            IF NO, GO ON
        B    @SELECT          IF YES, GO TO COLOR SELECT ROUTINE
NEXT4   CB   @FIRE,@H18       WAS FIRE BUTTON PRESSED?
        JNE  SKIP             IF NO, SKIP DRAW ROUTINE
*
* ROUTINE TO PLACE BLOCK ON SCREEN
*
DRAW    LI   R0,>300          VDP SPRITE ATTRIBUTE ADDRESS
        LI   R1,ROW           CPU BUFFER TO RECEIVE DATA
        LI   R2,2             FETCH 2 BYTES
        BLWP @VMBR            FETCH DOT ROW AND DOT COLUMN
        CLR  R7               INITIALIZE R7 AND R8
        CLR  R8               --FOR USE IN DIVIDE OPERATION
        CLR  R2               INITIALIZE OFFSET FOR PATRN ARRAY
        MOVB @ROW,R8          PUT DOT ROW IN R8
        SWPB R8               MAKE IT LEAST SIG BYTE
        AI   R8,9             ADD ROW OFFSET FOR COLOR BLOCK +1
        C    R8,@COLMAX       IS THE DOT ROW > 255?
        JLT  NOCORR           IF NOT, DO NOT APPLY CORRECTION
        S    @COLMAX,R8       IF SO, SUBTRACT 255
NOCORR  DIV  @D32,R7          DIVIDE DOT ROW OF BLOCK BY 32
        SLA  R7,8             CALCULATE BYTES IN PRECEEDING GROUPS
        A    R7,R2            ADD # OF BYTES IN PREVIOUS 32X8 BYTE GROUPS
        SRL  R8,2             DIVIDE REMAINDER BY 4
        A    R8,R2            ADD # BYTES ABOVE IN CURRENT 8 BYTE SET
        CLR  R7               INITIALIZE R7 AND R8
        CLR  R8               --FOR USE IN DIVIDE OPERATION
        MOVB @COL,R8          PUT DOT COLUMN IN R8
        SWPB R8               MAKE IT LEAST SIG BYTE
        AI   R8,8             ADD COLUMN OFFSET FOR COLOR BLOCK
        C    R8,@COLMAX       IS THE DOT COLUMN > 255?
        JLT  NOCORC           IF NOT, DO NOT APPLY CORRECTION
        S    @COLMAX,R8       IF SO, SUBTRACT 256
NOCORC  DIV  @D8,R7           DIVIDE BY 8
        SLA  R7.3             CALCULATE BYTES IN PRECEEDING 8 BYTE SETS
        A    R7,R2            ADD # BYTES IN PREVIOUS 8 BYTE SETS, THIS GROUP
        MOV  R2,R2            CHECK IF INSIDE PATTERN ARRAY  N
        JLT  SKIP             IF NOT SKIP SCREEN PLACEMENT
        C    R2,@MAX          CHECK IF INSIDE PATTERN ARRAY EEN
        JGT  SKIP             IF NOT SKIP SCREEN PLACEMENT
        LI   R0,>14           REPEAT SUBROUTINE CHECKS 20 TIMES
        BL   @CHECKS          BRANCH TO SUBROUTINE CHECKS
        CLR  R1               INITIALIZE R1 FOR BLOCK COLOR
        MOVB @FCOLOR,R1       STORE COLOR IN R1
        SWPB R1               MAKE IT LEAST SIG BYTE
        CLR  R0               INITIALIZE R0 FOR CURRENT ARRAY ELEMENT
        MOVB @PATRN(2),R0     COPY ARRAY ELEMENT AT OFFSET INTO R0
        SRL  R8,2             CALCULATE WHETHER BLOCK IS LEFT OR RIGHT
        JEQ  MARK1            IF 0 LEAVE BLOCK AS LEFT NYBBLE
        SRL  R1,4             IF 1 MAKE BLOCK RIGHT NYBBLE
        SWPB R0               MAKE CURRENT ELEMENT LEAST SIG BYTE
        SRL  R0,4             GET RID OF LEAST SIG NYBBLE
        SLA  R0,4             PUT REMAINING NYBBLE BACK
        JMP  MARK2            SKIP TO LABEL
MARK1   SLA  R0,4             GET RID OF MOST SIG NYBBLE
        SRL  R0,4             PUT BACK REMAINING NYBBLE
        SWPB R0               MAKE IT LEAST SIG BYTE
MARK2   A    R1,R0            ADD NEW COLOR TO ADJACENT VALUE
        SWPB R0               MAKE IT MOST SIG BYTE
        MOVB R0,@PATRN(2)     MOVE IT TO ARRAY AT OFFSET
        LI   R0,>0800         VDP PATTERN TABLE ADDRESS
        LI   R1,PATRN         CPU BUFFER
        LI   R2,>600          1536 BYTES TO MOVE
        BLWP @VMBW            WRITE TO REDRAW SCREEN
```

Listing 1 *Magic Crayon* continued

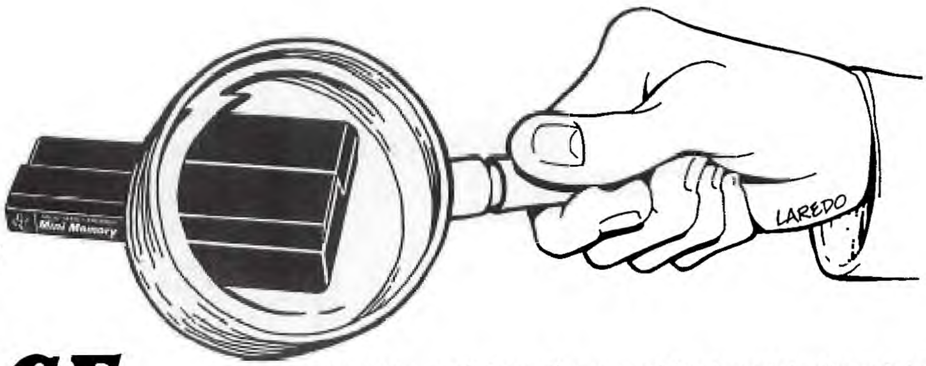
```
SKIP    CLR   R5               CLEAR R5 AND R6 TO RECEIVE JOYST VALUES
        CLR   R6
        MOVB  @JOYSTY,R5       PUT Y RETURN IN R5
        NEG   R5               MULTIPLY BY -1
        SLA   R5,2             MULTIPLY BY 4
        MOVB  @JOYSTX,R6       PUT X RETURN IN R6
        SLA   R6,2             MULTIPLY TIMES 4
        SWPB  R6               MAKE XVEL LEAST SIG BYTE
        MOVB  R5,R6            MOVE YVEL TO R6 AS MOST SIG BYTE
        LI    R1,USRWS+12      CPU ADDRESS OF VELOCITY BYTES (R6)
        LI    R0,>0780         VDP ADDRESS OF MOTION TABLE
        LI    R2,2             2 BYTES TO MOVE
        BLWP  @VMBW            WRITE DATA TO VDP RAM
        B     @CHECK            START LOOP OVER AGAIN
*
* ----- END OF MAIN PROGRAM LOOP --------------------
*
* COLOR SELECT ROUTINE
*
SELECT  LI    R0,>07EE         CHANGE BACKGROUND TO GRAY
        BLWP  @VWTR            WRITE TO VDP R7
        LI    R0,>800          VDP BUFFER FOR PATTERN TABLE
        LI    R1,PALET         CPU BUFFER FOR PALETTE
        LI    R2,>600          1536 BYTES TO MOVE
        BLWP  @VMBW            DISPLAY PALETTE
        LI    R0,>300          VDP BUFFER FOR ATTRIBUTE LIST
        LI    R1,ARRATT        ARROW ATTRIBUTES
        LI    R2,4             4 BYTES TO MOVE
        BLWP  @VMBW            WRITE DATA
        BL    @DEBNC           BRANCH TO "DEBOUNCE" SUBROUTINE
LOOP9   LIMI  2                ENABLE VDP INTERRUPT
        LIMI  0                DISABLE INTERRUPT
        MOVB  @ONE,@UNIT       IDENTIFY REMOTE UNIT TO SCAN
        BLWP  @KSCAN           SCAN LEFT KBD AND REMOTE UNIT #1
        CB    @FIRE,@H18       CHECK FIRE BUTTON
        JEQ   CMARK            IF PRESSED, CHANGE MARK COLOR
        CB    @FIRE,@H14       CHECK "C" KEY
        JEQ   CSCRN            IF PRESSED, CHANGE SCREEN COLOR
        CLR   R6               INITIALIZE R6
        MOVB  @JOYSTX,R6       PUT JOYST X IN R6
        SLA   R6,2             MPY BY 4
        SWPB  R6               MAKE LEAST SIG BYTE
        LI    R1,USRWS+12      LOAD CPU ADDRESS (R6)
        LI    R0,>0780         LOAD ADDRESS OF MOTION TABLE
        LI    R2,2             MOVE 2 BYTES
        BLWP  @VMBW            LOAD DATA TO VDP RAM
        JMP   LOOP9            GOTO LOOP9
CSCRN   BL    @DOTCOL          DETERMINE COLOR FROM DOT COLUMN OF ARROW
        SWPB  R1               MAKE IT MOST SIG BYTE
        MOVB  R1,@BCOLOR       MOVE IT TO BCOLOR
        JMP   BACK             JUMP TO BACK
CMARK   BL    @DOTCOL          DETERMINE COLOR FROM DOT COLUMN OF ARROW
        SLA   R1,12            PUT IN PROPER POSITION FOR @FCOLOR
        MOVB  R1,@FCOLOR       MOVE IT TO FCOLOR
BACK    BL    @DEBNC           DEBOUNCE
        CLR   R0               PREPARE TO RETURN SCREEN COLOR
        MOVB  @BCOLOR,R0       PUT BACKGROUND COLOR IN R0
        SWPB  R0               MAKE IT LEAST SIG BYTE
        MOVB  @H07,R0          INDICATE WRITE TO VDP R7
        BLWP  @VWTR            WRITE IT TO R7
        LI    R0,>800          VDP PATTERN TABLE ADDRESS
        LI    R1,PATRN         PATTERN BUFFER IN CPU RAM
        LI    R2,>600          1536 BYTES TO WRITE
        BLWP  @VMBW            LOAD PATTERN SCREEN
        LI    R0,>300          VDP SPRITE ATTRIBUTE TABLE ADDRESS
        LI    R1,ATTRIB        ADDRESS OF CURSOR ATTRIBUTES
        LI    R2,4             4 BYTES TO MOVE
        BLWP  @VMBW            LOAD DATA TO GET CURSOR SPRITE
        B     @SKIP            BRANCH TO LABLE SKIP
*
* DSR ROUTINE TO SAVE "SCREEN" -- PATTERN TABLE
*
SAVE    LI    R0,>1000         PREPARE TO MOVE PATRN TO VDP BUFFER
        LI    R1,PATRN         CPU BUFFER ADDRESS
        LI    R2,>600          1536 BYTES TO MOVE
        BLWP  @VMBW            WRITE DATA
        LI    R0,PAB           VDP PERIPHERAL ACCESS BLOCK ADDRESS
        LI    R1,PDATA         CPU BUFFER TO BE WRITTEN TO VDP
        LI    R2,21            21 BYTES TO WRITE
        BLWP  @VMBW            WRITE PAB
        LI    R6,PAB+9         SET POINTER TO NAME LENGTH
        MOV   R6,@PNTR         STORE IN >8356 >8357
        BLWP  @DSRLNK          EXECUTE SAVE OR LOAD
```

Listing 1 *Magic Crayon* continued

```
       DATA 8
       B    @CHECK             IF SO, BRANCH BACK TO BEGINNING
*
* DSR ROUTINE TO RECALL "SCREEN" -- PATTERN TABLE
*
RECALL LI   R0,PAB             VDP PERIPHERAL ACCESS BLOCK ADDRESS
       LI   R1,PDATA           CPU BUFFER TO WRITE
       LI   R2,21              21 BYTES TO WRITE
       BLWP @VMBW              WRITE PAB
       LI   R0,PAB             SUBSTITUTE "LOAD" I/O OP CODE
       MOVB @LOAD,R1           MOVE OP CODE TO R1
       BLWP @VSBW              WRITE BYTE TO PAB
       LI   R6,PAB+9           SET POINTER TO NAME LENGTH
       MOV  R6,@PNTR           STORE IN >8356 >8357
       BLWP @DSRLNK            COPY DATA TO VDP BUFFER
       DATA 8
       LI   R0,>1000           PREPARE TO COPY FROM VDP TO PATRN
       LI   R1,PATRN           CPU BUFFER ADDRESS
       LI   R2,>600            1536 BYTES TO COPY
       BLWP @VMBR              COPY BUFFER
       LI   R0,>0800           NOW COPY TO PATTERN TABLE
       LI   R1,PATRN           ADDRESS OF CPU BUFFER
       LI   R2,>600            1536 BYTES TO COPY
       BLWP @VMBW              COPY TO TABLE
       B    @CHECK             BACK TO THE BEGINNING
*
* SUBROUTINE TO PERIODICALLY CHANGE SPRITE COLORS
*
CHECKS AI   R4,>100            ADD 256 TO R4
       JEQ  CHANGE             WHEN R4 REACHES 0, CHANGE COLOR
       DEC  R0                 DEC COUNTER
       JNE  CHECKS             IF NOT 0 ADD ANOTHER 256
       JMP  RETURN             BACK TO MAIN PROGRAM
CHANGE SWPB R3                 SWITCH COLOR BYTES IN R3
       MOV  R3,R1              PUT R3 IN R1
       LI   R0,>303            ADDRESS OF SPRITE #0 COLOR IN VDP RAM
       BLWP @VSBW              WRITE MOST SIG BYTE OF R1
RETURN RT                      BACK TO MAIN PROGRAM
*
* DEBOUNCE SUBROUTINE
*
DEBNC  MOVB @ONE,@UNIT         KEY UNIT TO CHECK
       BLWP @KSCAN             SCAN KEYBOARD
       CB   @FIRE,@NOKEY       IS NO KEY PRESSED?
       JNE  DEBNC              IF A KEY IS PRESSED, CHECK AGAIN.
       RT                      GO BACK TO MAIN PROGRAM
*
* SUBROUTINE TO DETERMINE COLOR FOR ARROW
*
DOTCOL CLR  R1                 INITIALIZE R1 TO RECEIVE DOT COLUMN
       LI   R0,>301            VDP ADDRESS OF DOT COLUMN
       BLWP @VSBR              READ BYTE FROM ATTRIBUTE TABLE
       SWPB R1                 MAKE IT LEAST SIG BYTE
       AI   R1,>07             ADD OFFSET FOR POINT OF ARROW
       SRL  R1,4               DIVIDE BY 16
       RT                      RETURN
*
* "END START"
*
AUTO   END  MARKER             AUTOSTART
```

MINI MEMORY CARTRIDGE

There's More There Than Meets the Eye



You know, looks can be deceiving. Who'd suspect that a bespectacled, mild-mannered reporter for the *Daily Planet* could leap over tall buildings with a single bound? In the same way, there's more to TI's Mini Memory Command Cartridge than meets the eye. What appears to be a normal, garden-variety Command Cartridge, however, really converts your TI Home Computer from a good BASIC machine to a trim and efficient assembly language instrument.

Even the name is a clever disguise: "Mini" Memory, indeed! If you believe that there's just a tiny bit of memory in there, you probably believe that the Trojan Horse was nothing more than an overgrown hobby-horse! This cartridge actually has 14K bytes of memory: 4K of RAM, 4K of ROM and 6K of GROM.

RAM (read/write) memory is used by your computer to store your programs. And you know that any program you write disappears from the computer's memory when you shut the computer off. But *Mini Memory* has a surprise for you: When you shut the computer off and unplug the cartridge, your programs don't disappear from the cartridge's RAM. A battery inside the cartridge feeds a trickle of current to the CMOS devices—which are real power misers—and keeps them alive. And now you can carry your programs around with you, plug them in, and *instantly* load them—no cassettes, no diskettes, no messy cables, no long waits.

But there's more yet. Besides battery-backed RAM, this cartridge also has 4K bytes of ROM (Read-Only Memory) and 6K bytes of GROM (Graphics Read-Only Memory). The ROM and GROM give you seven additional TI BASIC subprograms, as well as access to many system routines from assembly language programs. The ROM also contains a powerful program debugger, EASY BUG, which can help you exterminate those pesky "logic vermin" which infest programs.

At this point, you may be saying to yourself, "What good does all this Assembly Language access and debugging stuff do for me, anyway, without an assembler?" Glad you asked. The Mini Memory Command Cartridge comes with an assembler on cassette. You can load this assembler into memory, enter assembly language statements, and have the assembler translate them into TMS9900 object code.

Let's explore this cornucopia one item at a time.

FILE STORAGE

Probably most persons will use the Mini Memory cartridge most often for temporary storage of programs and data. You can think of the Mini Memory cartridge as a very fast-access storage device. [See "Getting Down to Business" for a tutorial on random access files.—Ed.]

When you have the *Mini Memory* Command cartridge plugged in, the 4K-byte RAM has the file name MINIMEM for TI BASIC program and data storage. The RAM occupies physical addresses 28672 through 32767 (hexadecimal 7000 through hexadecimal 7FFF). You can save programs in this file and load programs from it. (For example, to save a TI BASIC program, just enter the command SAVE MINIMEM.) You can also store data in this file using the file specification available for any TI BASIC file. For example, the following statements open the Mini Memory file and store data values in the file.

```
OPEN #3 :"MINIMEM",RELATIVE,FIXED,
UPDATE,INTERNAL
PRINT #3: A,B,C,D
```

With the Mini Memory cartridge you can also access a second new file. EXPMEM2 is the name of a 24K-byte memory file located in the 32K Memory Expansion unit. EXPMEM2 is available, however, only if you have the Memory Expansion unit connected to your computer and turned on.

ADDITIONAL TI BASIC SUBPROGRAMS

Seven additional TI BASIC subprograms are yours with the Mini Memory cartridge. These subprograms are PEEK, PEEKV, POKEV, CHARPAT, INIT, LOAD, and LINK.

The PEEK subprogram reads bytes of CPU RAM data and copies the data directly into TI BASIC variables. For example, the statement:

```
CALL PEEK (8192,A,B,C,(8))
```

reads three bytes of data starting at address 8192, and assigns the values read to the variables A, B, and C(8).

The PEEKV subprogram reads bytes from VDP RAM. It works exactly like PEEK, except PEEKV accesses VDP RAM instead of CPU RAM.

The POKEV subprogram stores data values into VDP RAM. For example,

```
CALL POKEV(784,30,30,30)
```

writes the value 30 to VDP RAM locations 784, 785, and 786.

The CHARPAT subprogram reads a 16-character pattern identifier that specifies the pattern of a character code. For example,

CALL CHARPAT(68,D$)

places the pattern defining character code 68 in the string variable D$.

The three TI BASIC subprograms INIT, LOAD, and LINK interface Assembly Language programs and TI BASIC programs.

The INIT subprogram initializes the CPU memory for Assembly Language programs. The LOAD subprogram loads Assembly Language object files into CPU memory and it loads data into the CPU memory.

There are two forms of the LOAD subprogram. One form is used to load an object file from a storage device into memory, and the second form is used to load data directly into CPU memory. For example, the statement

CALL LOAD ("DSK1.DEMO")

loads the file DEMO from the diskette in Disk Drive 1.

The second form of the LOAD subprogram is a POKE function for CPU RAM. For example, the statement

CALL LOAD (8197,85,40)

loads the value 85 into memory location 8197 and the value 40 into memory location 8198.

The LINK subprogram passes control and, optionally, a list of parameters from a TI BASIC program to an Assembly Language program. For example, the statement

CALL LINK ("PROG1",A,E(9))

passes control from a TI BASIC program to an Assembly Language program named PROG1 and passes the variables A and E(9) to the program.

ACCESS TO SYSTEM ROUTINES

The utility routines resident in the Mini Memory Command Cartridge can be called from an Assembly Language program to access machine resources and interface with the TI BASIC interpreter. It's fair to warn you that the use of these routines requires a knowledge of the routines themselves and the organization of data used by the routines. You can get additional information about these routines from the Editor/Assembler owner's manual (available separately).

Two types of access programs are resident in the Mini Memory Command Cartridge. One program contains a collection of system utilities with which to link to ROM/GROM routines, perform a keyboard scan, access the VDP, etc. The individual utility programs are classified as either *Standard Utility* programs or *Extended Utility* programs.

A second program contains TI BASIC interface utilities with which an Assembly Language program can access variables passed through a CALL LINK statement in a TI BASIC program. This program also contains an error-handling utility to return exceptions to a TI BASIC program.

STANDARD UTILITY PROGRAMS

The following standard system utilities become accessible with the *Mini Memory* Command Cartridge:

—VDP Single Byte Write—Write a single-byte value to a specified VDP RAM address.

—VDP Multiple Byte Write—Write multiple bytes from CPU RAM to VDP RAM.

—VDP Single Byte Read—Read a single byte from a specified VDP RAM address.

—VDP Multiple Byte Read—Read multiple bytes from VDP RAM into CPU RAM.

—VDP Write to Register—Write single-byte value to any of the VDP RAM registers.

—Keyboard Scan—Scan the keyboard and return a key-code and status. This routine can also read the position of the Wired Remote Controller.

EXTENDED UTILITY PROGRAMS

Extended utilities are provided to access routines in the console GROMs and ROMs. These utilities are GPLLNK (link to GPL routines in GROM), XMLLNK (link to routines in ROM), and DSRLNK (link to Device Service Routines).

GPLLNK Routines

The GPLLNK routines are as follows:

—Load Standard Character Set—Load the standard character set into VDP RAM

—Load Small Character Set—Load the small character set (for the 40-column Text Mode) into VDP RAM.

—Execute Power-Up Routine—Initialize the system as if the computer had just been turned on.

—Accept Tone—Issue an accepting tone for input.

—Bad Response Tone—Issue a bad-response tone warning.

—Bit Reversal Routine—Provide a mirror image of a byte of information.

—Cassette Device Service Routine—Access a cassette tape recorder/player as a storage device.

—Load Lower Case Character Set—Load the lower-case character set into VDP RAM.

The following floating point routines are also available through GPLLNK:

—Convert a floating-point number to an ASCII string.

—Compute the greatest integer contained in a value.

—Raise a number to a specified power.

—Compute the square root of a number.

—Compute the inverse natural logarithm of a value.

—Compute the natural log of a number.

—Compute the cosine of a number.

—Compute the sine of a number.

—Compute the tangent of a number.

—Compute the arctangent of a number.

XMLLNK Routines

Routines in the console ROM can be accessed through the XMLLNK routine, The following routines can be called from an Assembly Language program using XMLLNK:

—Floating-point addition.

—Floating-point subtraction.

—Floating-point mutiplication.

—Floating-point division.

—Floating-point compare.

—Floating-point stack addition.

—Floating-point stack subtraction.

—Floating-point stack multiplication.

—Floating-point stack division.

—Floating-point stack compare.

—Convert a string to a number.
—Convert a floating-point format number to an integer.
—Push a value onto the value stack.
—Pop a value from the value stack.
—Convert an integer number to floating-point format.

DSRLNK Routines

DSRLNK links an Assembly Language program to a Device Service Routine (DSR) or a subprogram in ROM. As with GPLLNK and XMLLNK, TI cautions you to make sure you know what you are doing before using DSRLNK. [A DSR is a machine language program that TI has burned into ROMs found in each of its peripherals. Since each peripheral contains its own custom "operating system," the TI-99/4A did not have to be designed to anticipate future peripheral requirements.—Ed.]

TI BASIC INTERFACE UTILITIES

TI BASIC interface utilities allow an Assembly Language program to read or assign values to variables passed in a parameter list from a CALL LINK statement in a TI BASIC program. These utility routines include argument-passing utilities and an error-reporting utility.

The following are the TI BASIC interface utilities:
—Assign a numeric value to a numeric variable.
—Assign a string to a string variable.
—Retrieve the value of a numeric parameter.
—Retrieve the value of a string parameter.
—Report an error. (The Assembly Language program can report any existing TI BASIC error or warning message upon returning to TI BASIC.)

EASY BUG DEBUGGER

Also inside the *Mini Memory* cartridge's ROM is EASY BUG. EASY BUG is a versatile program development tool with which you can (1) debug your Assembly Language programs, (2) access the input/output ports of the computer, (3) load programs, and (4) store programs. And it really is easy to use. With EASY BUG, you can inspect and (optionally) modify the contents of CPU and VDP memory, display the contents of ROM, run Assembly Language programs from EASY BUG, directly access the peripheral devices which are connected to the computer via the 9900 microprocessor's serial I/O port (the CRU), and save or load programs on cassette.

LINE-BY-LINE SYMBOLIC ASSEMBLER

A line-by-line symbolic assembler on a cassette tape is supplied with the Mini Memory cartridge. It assembles Assembly Language statements and stores the object code directly into the 99/4A's CPU RAM. You can make both forward and backward references to one- or two-character labels with the Assembler. Each source statement you enter is immediately assembled into object code and stored into memory. Because some source code is retained in a nine-page text buffer, you can scroll the screen to review previously entered lines of source code by pressing the up- and down-arrow keys. The source program cannot be saved, however.

The Line-by-Line Assembler occupies about 2K bytes. When it is loaded into the Mini Memory cartridge's 4K byte RAM, you still have about 2K bytes of memory for your Assembly Language program.

Assembler Directives

The Assembler recognizes seven directives:
—The AORG (Absolute Origin) directive establishes the location counter value to set the starting address of assembled code.
—The BSS (Block Starting with Symbol) directive reserves a block of initialized memory.
—The DATA (Data Initialization) directive initializes a word or words of memory to a specific value.
—The END (End Program) directive terminates the assembler and causes a display of the number of unresolved references, if any.
—The EQU (Equate) directive defines a value for a symbolic constant.
—The SYM (Symbol Table Display) causes a display of all symbols and their values in the program.
—The TEXT (String Definition) directive causes a string of characters to be translated into their ASCII code and stored as a part of a program.

[Rather than being strictly a part of the internal logic of your program, assembler directives are commands which direct the Assembler to perform certain operations at assembly time.—Ed.]

DEMONSTRATION PROGRAM

Along with the Line-by-Line Assembler on the cassette is an Assembly Language demonstration program called LINES which draws a colorful line design on the screen. The LINES program can be run only on the TI-99/4A Home Computer, however, because it requires the enhanced graphic processor contained on the TI-99/4A.

OPERATION

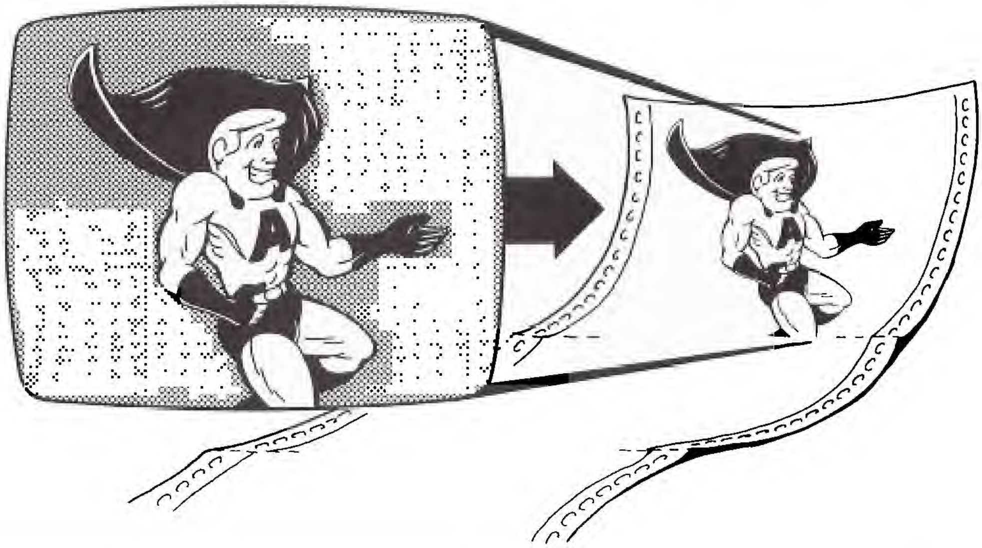
TI has a knack for creating complex and versatile programs that are still simple to operate; they've definitely done it again with the Mini Memory Command Cartridge. When you plug in the cartridge, turn on the computer, and pass the opening credits on the Master Title Screen, you are presented with a simple, three-choice selection screen. You can choose TI BASIC, EASY BUG, or MINI MEMORY.

If you select MINI MEMORY, you are presented with a second three-choice selection screen. You can choose to load an object program into memory and run it, run a previously loaded program already in memory, or reinitialize the cartridge to prepare it for loading new programs or storing data. Pick a number, pluck a key, and you're off and running. It's as easy as eating oatmeal cookies!

CONCLUSION

This has got to be one of the best deals around. 4K bytes of RAM with battery backup assure that all the good stuff stored in the RAM is not lost when you turn off the console or even when you remove the cartridge. 10K bytes of ROM and GROM give you seven additional TI BASIC subprograms (including PEEK and POKE), access to system routines from Assembly Language, and routines to allow you to interface Assembly Language programs to TI BASIC. You've got a user-friendly program debugger, a symbolic line-by-line assembler, and a captivating graphics demonstration program. All of this, plus 84 pages of documentation, for $99.95 (suggested retail price). With all this to offer, it's really not too hard to see why there's definitely more to the Mini Memory Command Cartridge than meets the T-eye .

A Screen Printing Utility



PART 1: Design Considerations

One of the best features of the TI-99/4A computer is its graphics capability. The programmer can create a huge variety of screens by using the simple character-definition commands of TI BASIC. Wouldn't it be nice to dump those screens to your non-thermal printer? This two-part article presents a method for doing this on the TI-99/4 impact printer. Part I discusses the theory behind the screen dump. Part II will provide the Assembly Language subroutine itself.

I should mention that the 99/4A has an improved video processor (TMS9918A) which allows you to define up to 768 unique characters on the screen. However, this bit-map mode requires an extra 12K of memory to hold the larger tables needed. We'll limit ourselves to the Graphics I, or standard mode, in this discussion.

Approach—in English

The video screen contains 768 character positions, arranged in 24 rows of 32 characters. Each character is composed of an 8×8 dot matrix, giving you a screen of 192 × 256 dots. The screen dump program will reproduce the screen dot-for-dot on the printer.

With bit-image mode selected, the TI-99/4A prints characters which are one dot wide and 8 dots high. Since the screen characters are also 8 dots high, each screen character can be represented by 8 TI-99/4A bit-image characters, for a total of 64 possible dots per screen character.

Accessing the Screen Image

The contents of the screen are stored in VDP RAM. Since we are not concerned with color here, only two of the screen tables in VDP RAM are of interest. The first is the Screen Image Table, which starts at default address>0000 and contains 768 bytes. Each byte corresponds to the character position on the screen and contains the character number occupying that screen position. VDP RAM addresses>0020 through >003F correspond to the second screen row, and so on. Since each character number is contained in one byte, you can see that the character numbers must be between >00 and >FF, or decimal 0 through 255.

The second table we'll need is the Pattern Descriptor Table, which starts at VDP RAM address>0800 by default. This table contains the dot patterns for each of the 256 characters which can be in use. The BASIC subprogram CHAR, which is used to define dot patterns for characters, stores patterns in this table. Since a character pattern takes 8 bytes to define, and there can be up to 256 different characters, the Pattern Descriptor Table occupies 2084 bytes of VDP RAM.

Figure 1 shows the relationship between these two tables. For a given screen ROW and COLUMN, the VDP RAM address of the corresponding character number is given by (ROW − 1) * 32 + COLUMN − 1. Once you have obtained this character number, you can use it to index to the correct spot in the pattern Descriptor Table. The offset in this table is just 1024+(N−32)*8 in decimal, since each pattern description is 8 bytes long. Figure 1 shows an example of finding the pattern for the home position (ROW 1, COLUMN 1) on the screen. The character number resides in the Screen Image Table at address 0. If the home character on the screen is "A", then VDP RAM address 0 contains the value 65 or >41. From the offset in the Pattern Descriptor Table, we get VDP RAM address>800 + >200 = >0A00. The eight bytes starting at >0A00 in VDP RAM contain the pattern for the character "A". You can see that for our purposes, the contents of the Screen Image Table are just intermediate, though necessary, data. The character pattern is what we're really after.

The 8-byte character pattern represents the dot pattern which appears on the screen in what I'll call *row-wise* form. The top portion of Figure 2 illustrates this for the character "A". The first byte of the pattern represents the first row of the dots which comprise the character. The hexadecimal notation is just a shorthand way to group four bits at a time, with bits of value 1 standing for dots which are turned on in the character.

Translating the Characters to TI-99/4A Format

The TI-99/4 printer constructs its bit-image output in a different way. It uses what I'll call *column-wise* form. It still takes 8 bytes to produce the same character, but each byte of data passed to the printer represents a column (rather than a row) of dots in the finished character. The bottom of Figure 2 illustrates this. If we think of the character's dot pattern as an 8×8 matrix, then the translation from TI internal format to TI-99/4A printer bit-image format is equivalent to transposing the matrix. We can't really treat each character pattern as a 64-bit matrix because 9900 Assembly Language does not have a BIT data type, but we can base the logic of the program on this idea.

Figure 1.

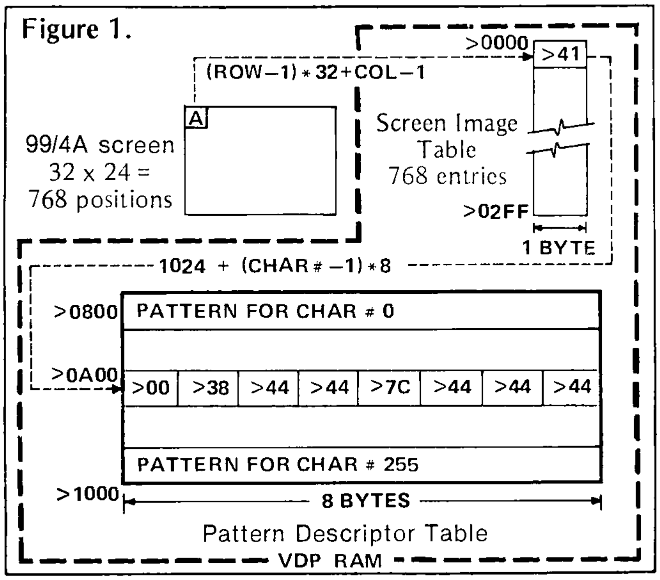


Figure 2.

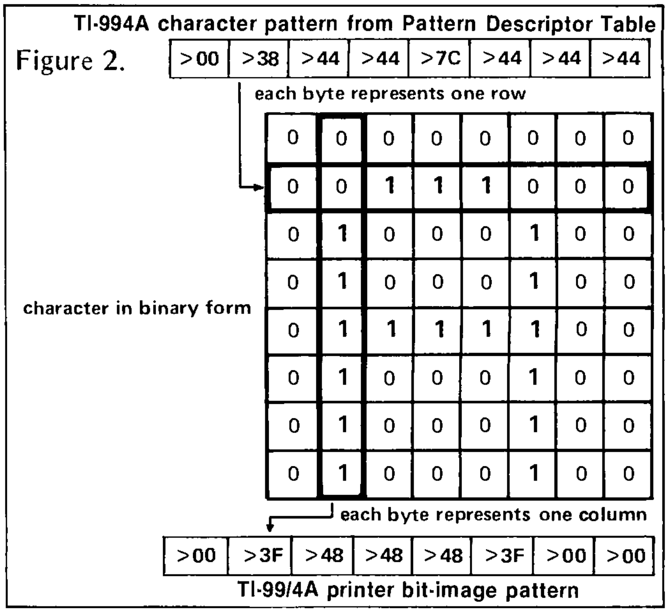
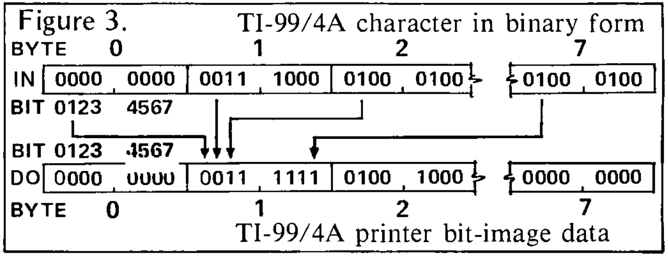


Figure 3.



Program Outline

The screen dump program reads the Screen Image Table one byte at a time starting at the top (VDP RAM address 0). The value of each byte is used to calculate the position of the character pattern, and the 8-byte pattern is obtained from the Pattern Descriptor Table. These 8 bytes will be manipulated to produce 8 bytes of information encoded for the TI-99/4 printer. Figure 3 shows how the bits of the TI-99/4A character pattern are rearranged to form bit-image data for the printer. Notice that the data at byte M, bit N is moved to byte N bit M—or transposed. The program will also have to send certain control characters for bit-image mode to the printer.

PART 2: Screen Dump

The Assembly Language subroutine for dumping 99/4 screens to the TI-99/4 impact printer is designed to be called from console BASIC, and can be entered into your system using either the Editor/Assembler or the Line-by-Line Assembler in the Mini Memory Command Cartridge.

VDP RAM Under Console BASIC

When the TI-99/4A is under control of the BASIC interpreter, VDP RAM contains two areas of interest here. VDP RAM addresses >0000 – >02FF (0 – 767 in decimal) contain the character numbers associated with each screen position. The character patterns for character numbers 32 – 159 start at VDP RAM address >0400 (1024). In the Pattern Descriptor Table address the 8-byte character pattern corresponding to a character number N is 1024 + (N – 32) * 8 in decimal.

The dump subroutine (called DUMP) uses these facts. Starting with VDP RAM address 0, DUMP gets the screen character number and uses it to calculate the VDP RAM address of the associated character pattern. It then reads the 8-byte character pattern, transposes the matrix, and writes the resulting 8 bytes to the printer. DUMP performs this process on each successive byte of screen RAM, up to and including VDP RAM address >02FF (767).

DSRLNK and Printer Output

The actual output to the printer is done by means of a built-in Extended Utility Routine called DSRLNK.

Before calling DSRLNK, the Assembly Language subroutine must set up a Peripheral Access Block (PAB) in VDP RAM. Here is the format of the PAB we'll use for the printer:

BYTE#	CONTENTS
0	I/0 opcode: >00 = open >01 = close >03 = write
1	Flag/status byte. >12 is the code for sequential file, output operation, DISPLAY type data and variable length records.
2, 3	Data buffer address in VDP RAM. We'll use >1E00.
4	Logical record length.
5	Number of characters to write.
6, 7, 8	Not used here.
9	Length of file descriptor which follows.
10–35	File descriptor. We'll use RS232.PA=O. DA=8.BA=9600.CR

We'll put the PAB in VDP RAM starting at address >1D00 (hereafter called V1D00), and we'll put the data area containing the actual data for output to the printer at V1E00. These addresses could have been elsewhere in VDP RAM, as long as the locations chosen were not used by something else.

To perform a printer operation, the program must do the following:

1. Build the PAB in VDP RAM.
2. Put the address of the length of the file descriptor (byte 9 of the PAB) into CPU RAM address >8356.
3. Call DSRLNK.

You'll notice that the call to DSRLNK must be followed by a word (two bytes) containing the value 8, which means that you want to link to a Device Service Routine (DSR).

RS232 Considerations

Since the DUMP subroutine uses the RS232 interface to communicate with the printer, some additional code is needed to save and restore the address of the GROM. This is because the GROM address is changed when the RS232 DSR is used. At the beginning of the DUMP subroutine, the GROM address is obtained one byte at a time from the GROM Read Address at location >9802. The GROM address increments itself when the first byte is read (actually moved) from the GROM Read Address. This makes the second byte of the GROM address one too big, so it must be decremented by DUMP. Just before returning to BASIC, the DUMP subroutine restores the GROM address by moving it to the GROM Write Address at location >9C02, again one byte at a time.

Linkage to Console BASIC

A console BASIC program invokes the DUMP subroutine by the statement CALL LINK("DUMP"). DUMP returns to the BASIC program by branching to the contents of register 11 (R11). Just before returning to BASIC, the DUMP subroutine clears the error byte at @>837C (sets it to 0). Failure to clear this byte can result in an undeserved INCORRECT STATEMENT error when you return to BASIC.

Transposing the 8x8 Character Matrix

Once a screen character's 8-byte pattern has been read into CPU RAM (at label IN), the DUMP subroutine uses the following technique to build the 8 bytes of output at label DO.

The first byte of DO is composed of the first bit of each of the 8 bytes starting at IN, the second byte of DO is composed of each second bit of the bytes at IN, and so on. Figure 2 of Part One shows the bit movements for the pattern character of an "A".

DO is built from left to right, and R4 is used to hold each byte of DO as it is built. R4 is cleared before each byte is built, so DUMP has to turn on any bits necessary.

To tell if a certain bit of IN is on, DUMP compares the value of the byte containing the bit in question to a power of 2. To see how this works, consider the byte containing >82 (130 in decimal, 1000 0010 in binary). The leftmost bit of the byte is on; in fact, the leftmost bit would be on in any byte containing >80 (128) through >FF (255). In other words, we could test for the leftmost bit's being on by comparing the value of the byte to decimal 128 (2 to the 7th power); if the value is less than 128, we wouldn't have to turn on the corresponding output bit.

This technique can be used to test any bit of a byte for our purposes, using the appropriate power of 2. The second-to-leftmost bit can be tested against 64, its neighbor to the right against 32, and so on down to 1 for the rightmost bit. This works because we'll be considering the bits from left to right in each byte. After each bit is tested, it must be turned off (in CPU RAM, not on the screen) so that it doesn't interfere with the test of the following bit. To see this, consider the byte containing >82 (130) again. If we want to determine if the second-to-leftmost bit is on, we would compare the byte to 64. You can see that it would pass the test, even though the bit in question is not on! However, if we had reset the leftmost bit to 0 after testing it previously, the byte would now contain >02 instead of >82, and the test would fail, as it should.

Once we have decided that an input bit is on, we must set on the corresponding bit in R4. This is done by adding the appropriate power of 2 to R4. To turn on the leftmost bit, add 128; to turn on the rightmost bit, add 1. Remember that the byte being built is in the right half (LSB, or least significant byte) of R4.

DUMP uses R5 to contain the power of 2 for testing whether the input bit is on, and R6 to contain the power of 2 for setting the bit on for output. The LSB of R7 contains the input byte being tested, and the most significant byte of R7 is filled with zeros. This allows DUMP to use the simpler and more plentiful register instructions, and to completely avoid having the leftmost bit of a byte interpreted as a sign bit.

Printer Consideration

DUMP writes one full screen line to the printer at a time. Before each line, the program must write a 4-byte control sequence to put the printer in graphics mode and tell it how many graphics characters are coming next. This sequence is >1B, >4B, >FF, and >00. The last two bytes mean 255 characters will be written, with the order of the bytes being reversed for evaluation (>00FF, or 255).

The program issues a carriage return and line feed only after each of these writes, that is, at the end of each screen line. DUMP uses the CZC (Compare Zeros Corresponding) instruction to accomplish this. R9 contains the position in VDP RAM of the next screen character number. Positions 0 – 31 (>00 – >1F) of VDP RAM correspond to the characters on line 1 of the screen; positions 32 – 63 (>20 – >3F) correspond to characters on line 2, etc. The CZC instruction occurs right after R9 is incremented and before the corresponding screen character is decoded. Therefore, the carriage return and line feed should be written whenever R9 is an even multiple of 32. The CZC instruction uses a mask of >1F (0000 0000 0001 1111 binary). If R9 is a multiple of 32, then its last five bits will all be zero. Notice that the mask has only the last five bits turned on. Under these circumstances, the CZC instruction sets the equal status bit on if and only if the last 5 bits of R9 are all zero, that is, if and only if R9 contains an even multiple of 32. The JNE instruction which follows the CZC instruction causes the program to skip outputting the carriage return and line feed when R9 does not contain a multiple of 32.

Left to its own devices, the printer will respond to a line feed by spacing down 1/8" or 1/6". This would leave blank stripes in the screen dump. The sequence ESCAPE A 8 is written by DUMP to tell the printer to space down only 8/72" on each line feed. This produces a continuous image.

Mini Memory Considerations

To enter the DUMP subroutine via the Line-by-Line Assembler, do the following:

1. Select MINI MEMORY and then RUN from the first two menus.
2. Enter NEW as the program name.
3. When the Line-by-Line Assembler screen appears, type a space, then AORG, another space, >7D14, and then press [ENTER.] (From now on the spaces will be assumed.) This sequence lets you start the program at >7D14 instead of the traditional >7D00.
4. Enter the program as shown in Listing #1. Enter only the label (if any), opcode, and operands. Don't enter END yet.
5. Put the entry point for DUMP into the DEF/REF table by entering the following lines:

```
AORG >7FE8(CR)
TEXT 'DUMP   '(CR)
DATA >7D14(CR)
```

6. Set the last used address in Mini Memory by entering:

```
AORG >701C(CR)
DATA >7F02(CR)
```

7. Indicate that you are finished by entering:

```
END(CR).
```

The system should show that you have no unresolved references. Press enter twice, and then QUIT the Line-by-Line Assembler.

8. Enter EASY BUG from the master menu.
9. Press any key to bypass the instruction screen.
10. Enter S7000 when the system prompts with ? and then 7FFF when the system prompts TO? This tells the system to save the contents of the Mini Memory to cassette tape. Just follow the instructions presented by the computer after this, and then QUIT EASY BUG when you have saved and checked your tape.

You are now ready to use the DUMP subroutine. The sample BASIC program in Listing #2 just draws a screen and then waits for you to press the P key, at which point DUMP is called to print out the screen. You can incorporate DUMP into your own programs in any way you choose. Happy dumping!

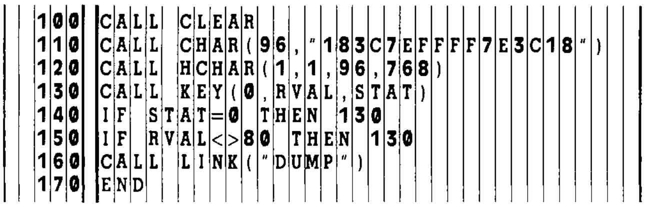
Listing 1 *Dump*

```
         AORG >7D14
         MOVB @>9802,@S1     GET MSB OF GROM ADDR INTO S1
         SWPB @S1
         MOVB @>9802,@S1     GET LSB OF GROM ADDR
         SWPB @S1
         DEC  @S1             CORRECT FOR AUTO-INCREMENT
         LI   0,>1D00
         LI   1,PD
         LI   2,36
         BLWP @>6028         WRITE PAB TO VDP RAM
         LI   6,>1D09
         MOV  6,@>8356       POINT TO DEVICE NAME LENGTH
         BLWP @>6038         DSRLNK TO OPEN PRINTER
         DATA 8
         LI   10,>0400
         MOV  10,@>7DEA
         LI   0,>1D00
         LI   1,>0300
         BLWP @>6024         PUT WRITE OP CODE IN PAB
         LI   0,>1D05
         LI   1,>0400
         BLWP @>6024         PUT LENGTH OF 4 IN PAB
         LI   0,>1E00
         LI   1,E2           PUT CODE FOR CARRIAGE RTN &
         LI   2,4            8/72" VERTICAL LINE SPACING
         BLWP @>6028         IN DATA BUFFER.
         MOV  6,@>8356       POINT TO DEVICE NAME LENGTH
         BLWP @>6038         DSRLNK-CHANGE VERT SPACING
         DATA 8
         LI   10,50          DELAY
         DEC  10
         JNE  $-2
         CLR  9              R9->NEXT SCREEN POSITION
L0       MOV  9,0
         BLWP @>602C         PUT BYTE OF SCREEN RAM IN R1
         SRL  1,8            SHIFT TO LSB OF R1
         AI   1,-128         ADJUST FOR BASIC
         SLA  1,3            *8
         AI   1,1024         PTRN ADDR=1024+(CHAR#-32)*8
         MOV  1,0
         LI   1,IN
         LI   2,8
         BLWP @>6030         PUT PATTERN INTO IN
         LI   5,128          R5 = BIT#
         CLR  8              R8 = OFFSET FOR DO
L3       LI   6,128          R6 =  BYTE#
         CLR  3              R3 = OFFSET FOR IN
         CLR  4              R4 IS FOR BUILDING NEXT CHAR
L2       CLR  7
         MOVB @IN(3),7       R7 HOLDS BYTE BEING DECODED
         SWPB 7              PUT BYTE IN LSB OF R7
         C    7,5            IS BIT ON?
         JLT  L1             NO
         A    6,4            YES,TURN OUTPUT BIT ON
         S    5,7            TURN OFF INPUT BIT
         SWPB 7              PUT BYTE IN MSB OF R7
         MOVB 7,@IN(3)       REWRITE TO IN
L1       INC  3              POINT TO NEXT INPUT BYTE
         SRA  6,1            /2
         JGT  L2             DO NEXT BYTE, IF MORE
         SWPB 4              PUT OUTPUT BYTE IN MSB OF R4
         MOVB 4,@DO(8)       STORE AT DO
         INC  8              POINT TO NEXT BYTE OF DO
         SRA  5,1            /2
         JGT  L3             CONSTRUCT NEXT OUTPUT BYTE
         LI   0,>1D05
         LI   1,>0000
         BLWP @>6024         PUT LENGTH OF 4 IN PAB
         LI   0,>1E00
         LI   1,E1
         LI   2,4
         BLWP @>6028         PUT ESC K SEQ. IN DATA BUFF
         LI   6,>1D09
         MOV  6,@>8356       POINT TO DEVICE NAME LENGTH
         BLWP @>6038         DSRLNK TO WRITE ESC K SEQ.
         DATA 8
         LI   10,>0000
         MOV  10,@>7DEA
         LI   0,>1D05
         LI   1,>0800
         BLWP @>6024         PUT LENGTH OF 8 IN PAB
         LI   0,>1E00
         LI   1,DO
```

Listing 1 *Dump* continued

```
           LI     2,8
           BLWP   @>6028          PUT DO INTO DATA BUFFER
           MOV    6,@>8356        POINT TO DEVICE NAME LENGTH
           BLWP   @>6038          DSRLNK TO OUTPUT 8 CHARS
           DATA   8
           LI     10,50           DELAY
           DEC    10
           JNE    $-2
           INC    9               POINT TO NEXT SCREEN POSITION
           CI     9,767           DONE WITH SCREEN YET?
           JGT    L4              YES
           CZC    @MK,9           NO.  ARE WE AT END OF LINE?
           JNE    L0              NO-DO NEXT  SCREEN CHARACTER
           LI     0,>1D05         YES-OPUTPUT CR LF
           LI     1,>0200
           BLWP   @>6024          PUT LENGTH OF 2 IN PAB
           LI     0,>1E00
           LI     1,CR
           LI     2,2
           BLWP   @>6028          PUT CR LF INTO DATA BUFFER
           MOV    6,@>8356        POINT TO DEVICE NAME LENGTH
           BLWP   @>6038          DSRLNK TO OUTPUT CR LF
           DATA   8
           LI     10,>0400
           MOV    10,@>7DEA
           JMP    L0              DO NEXT SCREEN CHARACTER
L4         LI     0,>1D00         COME HERE WHEN FINISHED DUMP
           LI     1,>0100
           BLWP   @>6024          PUT CLOSE OP CODE IN PAB
           MOV    6,@>8356        POINT TO DEVICE NAME LENGTH
           BLWP   @>6038          DSRLNK TO CLOSE PRINTER
           DATA   8
           LI     10,50           DELAY
           DEC    10
           JNE    $-2
           MOVB   @S1,@>9C02      RESTORE SAVED DATA TO GRMWA
           SWPB   @S1
           MOVB   @S1,@>9C02
           SB     @>837C,@>837C CLEAR ERROR BYTE FOR BASIC
           LI     10,50           DELAY
           DEC    10
           JNE    $-2
           B      *11             RETURN TO BASIC
IN         BSS    8               AREA FOR SCREEN PATTERN
DO         BSS    8               AREA  FOR PRINTER PATTERN
MK         DATA   >001F           MASK FOR EOL TEST
PD         DATA   >0012,>1E00,>FF00,>0000,>001A
*                                 PAB DEFINITION
           TEXT   'RS232.PA=O.DA=8.BA=9600.CR'
*                                 DEVICE NAME
CR         DATA   >0D0A           CR LF
E1         DATA   >1B4B,>FF00     ESC K GRAPHICS SEQUENCE
S1         BSS    2               SAVE AREA
E2         DATA   >0D1B,>4108     CR AND ESC A VERT SPACING
           END
```

Listing 2 *Screen Dump*



```
100 CALL CLEAR
110 CALL CHAR(96,"183C7EFFFF7E3C18")
120 CALL HCHAR(1,1,96,768)
130 CALL KEY(0,RVAL,STAT)
140 IF STAT=0 THEN 130
150 IF RVAL<>80 THEN 130
160 CALL LINK("DUMP")
170 END
```