==========

L.A. 99ers PRESENTS

==========

ASSEMBLY DIGEST

<* Feb. 1985 *>

****

A Collection of articles, tutorials and Programs

****

* review *

By Hector Santos

Review: Assembly Language Books
by Hector Santos

It is ironic that users of the TI-99/4A had to suffer a dearth of good books and software for their computer until Texas Instruments announced their withdrawal from the home computer market. These are now available like never before. Where we settled for less than the ideal, our problem, now, is in sifting through those that are available and selecting what we really need.

Such is the case with books relating to the TMS9900 Assembly Language. We used to have the Editor/Assembler Manual and the Software Development Handbook from Texas Instruments . We now have at least five other sources. We will cover three of them in this article and hope to cover the others in a subsequent article.

The three books are:

Introduction to Assembly Language for the TI Home Computer by Ralph Molesworth, Steve Davis Publishing
139 pages, softcover, $16.95

Learning TI-99/4A Home Computer Assembly Language Programming by Ira McComic, Pretice-Hall
331 pages, softcover, $16.95

Fundamentals of TI-99/4A Assembly Language
by M. S. Morley, Tab Books
210 pages, softcover, $11.50

Introduction to Assembly Language for the TI Home Computer, by Ralph Molesworth, was the first to come out. It attempts to teach assembly language by making comparisons to BASIC statements. Presumably, this helps you learn faster by letting you draw on your BASIC programming skill.

After an almost cursory explanation of assembly language, addressing formats, and the registers, you are thrown into programming. The instruction set is not covered separately, and detailed memory maps and appendices are not provided because the book was meant to be used with the Editor/Assembler Manual. At the end of the early chapters are references to pages in the Editor/Assembler Manual and words in the glossary that should be read. You learn assembly language by following the explanations provided in the illustrative programs. The programs cover Input/Output, file handling, and sorting and handling arrays. Of interest is Chapter 11, which discusses how to incorporate assembly language routines in your BASIC programs. The difference in coding for the Editor/Assembler and the Mini-Memory modules are clearly explained. Assembling and running programs using the two modules are also explained.

Unfortunately, the book was not provided with an index, a necessity in a book of this nature. Do not blame the author; it is usually the publisher's responsibility to provide an index.

As the title states, the book is an introduction . It shoul be enough to get you started in the right direction, but do not expect to learn advanced programming techniques like setting sprites in motion and checking for their coincidence.

The next book, Learning TI-99/4A Home Computer Assembly Language, by Ira McComic, is more formally organized. Although it has a tutorial quality in the way that you are guided through sample programs, its format is more like a college text. After a short introduction to assembly language, data structure, and the TI-99/4A architecture, an overview of the instruction set is presented with more detailed explanation later in the book. Two chapters are devoted to a discussion of the addressing formats with Examples to illustrate how each is used. Chapters 9 and 10 cover the use of the Editor, the Assembler, the Loader, and the Debugger utilities of the package.

The book was written more for use with the Editor/Assembler. Of course, the principles you learn are just as valid for the Mini-Memory or the p-System Assembler-Linker. A very good description of how assembly language is handled by the Mini-Memory module is presented in Chapter 18 as well as a description of the capabilities of the p-System Assembler-Linker.

Illustrations and tables are used throughout the book, helping you visualize the storage and movement of data and the manipulation of bits in the registers. These make it a lot easier to learn the subject than with text-only material.

Appendices are also included. The most useful is the alphabetical list of instruction code. Designed for quick ref-rence, detailed and concise information is provided for each instruction code. I believe that after you learn assembly language, this is one book that you will keep referring back to.

In contrast, Fundamentals of the TI-99/4A Assembly Language, by M.S. Morley, was written specifically for use with the Mini-Memory module. The author had in mind users who want to learn assembly language but who do not own an expansion system . The Mini-Memory , in conjunction with the line-by-line assembler , enables you to get started with very little investment.

Morley starts out almost like McComic, covering assembly language, the TI-99/4A architecture, addressing modes, and the instruction set. Then, the use of the Mini-Memory is explained. You are shown how to enter the source code, and how to assemble and debug the program.

At this point, McComic and Morley diverge in their treatment of the subject. McComic comes back and discusses the instruction set in detail, and uses programs to illustrate the topics . Morley devotes the last two-thirds of his book to routines. These may be incorporated in your own programs and are very thoroughly explained. The appendix summarizes all the operation codes, grouped by categories. While not as complete as McComic's book, it still is very useful.

We have three books, each differing from the others in its approach to teaching assembly language. What about the Editor/Assembler Manual ? I have heard a lot of people criticize the book, saying it is impossible to learn assembly language from it. The Manual is a reference book and was not meant to teach. It is a very good book: very thorough, very detailed in is coverage of the subject.

When we try to learn a new subject, we turn to tutorials for help. Once we learn it, tutorials become of les use and we turn to reference manuals for quick and concise information. I remember when I was trying to learn TI BASIC, I really liked Beginner's BASIC, Now I find the TI-99/4A User's Reference Guide more useful.

Assembly language is substantially more difficult than BASIC. I suspect that there can never be a book, tutorial or whatever, that will make it easy to learn.

EDITOR ASSEMBLER TUTORIAL: Part II by M. Baker

Welcome back to another blinding session of assembly language. Before we really get into this new language it is imperative that we get some basics out of the way. The predominant mode in assembly language is its use of numbers, naturally. Although decimal can be used for the most part an understanding of HEX is extremely desirable. Addresses for example are HEX numbers. To differentiate between decimal and HEXadecimal numbers we place the "greater than" symbol) before the number such as)768.

What is hexadecimal? First, what is decimal? Deci means ten and HEX means sixteen. The term refers to and dictates the "BASE" of the number system in use. The number of DIGITS in any number system always counts up to ONE LESS than the BASE. So, in our everyday number system of decimal the digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. There are 10 digits and we can count up to 9. The number TEN is not a digit but composed of two digits, '1' and '0'. Only single digit or character representations may be used. When we are converting from one base to another it may "appear" this is not true but that is not the case. Also, remember, counting starts from ZERO and not ONE!

Well we already said that hex means sixteen. Remember. That's two digits. One less than the base is fifteen and that too is two digits. The problem is in our thinking, not the numbers. We are thinking in base 10 of a number system higher than base 10. We don't have too much of a problem when we use base 2 (BINARY) or base 8 (OCTAL) since those are 'within' our present number usage. Since numbers are merely representations of quantities dictated by us we could actually use anything to represent a number system. How about hieroglyphics! Maybe Roman Numerals? As you probably already know these are representations that man has used. We don't want to overcomplicate things do we. We already have ten digits and all we need are six more. Why not letters? Ok. Which ones? In the world of mathematics someone a long time ago very arbitrarily set up that letters of the alphabet could indeed represent numbers. Not only that but letters in the beginning of the alphabet should generally represent unvarying quantities while those at the end represent changing values. That is variables or unknowns. The proverbial x. (Boy, I tell ya, someone always comes up with some kind of rule.) Let's do that them. The letters of course are A, B, C, D, E, AND F. Now we have sixteen 'digits' or characters to represent our hexadecimal base. In order now they are: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.

Just as 10 in decimal represents a 'jump' to the next 'unit' so does)10 in HEX. However their Values or weightings are very different.)10 is equivalent to 16 in decimal and 10 is equivalent to)A in HEX. Let's show a number line to get the 'feel' of all this:

```
d 0 1...9 10 11 12 13 14 15 16...25 26 27 28 29 30 31 32...
h 0 1...9  A  B  C  D  E  F 10...19 1A 1B 1C 1D 1E 1F 20...
```

Let's do some simple hex arithmetic. If we add)5 and)4 we come up with)9. Looks easy. That's the same answer as in decimal. But, if we add)5 and)5 we now get)A. How about)33 and)66. A nice)99. Everything works smoothly until the proverbial CARRY. It' not so bad but you do need 15 fingers. How about)43 and)66. What did you get? I hope it was)A9. I'm not going to get into conversion now since that will take an awful lot of space. The simplest way to attack this hex stuff is to use tables. You remember those. Just like the multiplication tables you were forced to learn as a child. After you get the 'feel' of hex with that you can do some reading (unfortunately on your own). You will then be better equiped to tackle conversions, etc.

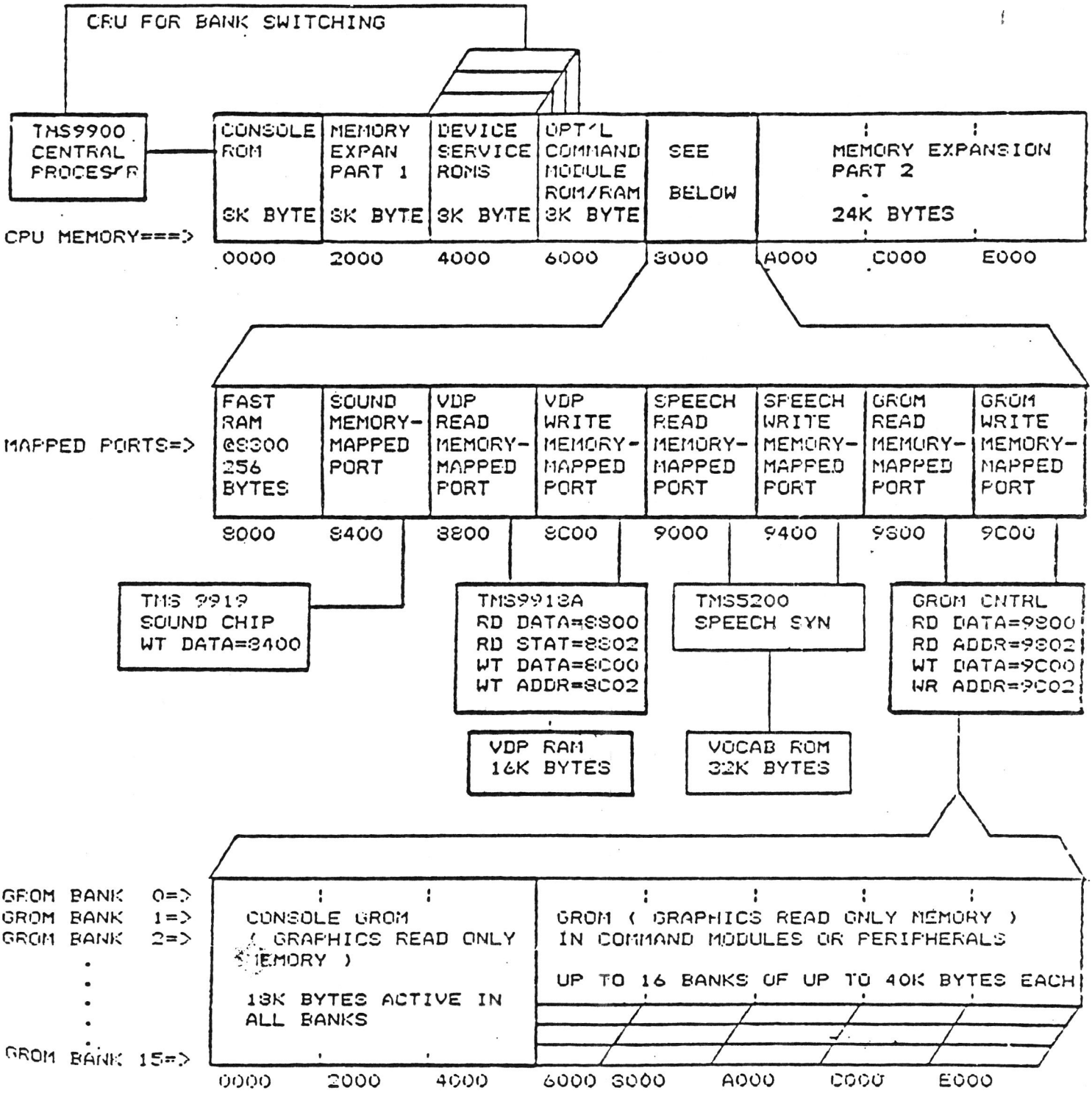
To get you on the move I've included the tables for addition and multiplication in hex. Good Luck, good reading (which you MUST do) and see you next newsletter.

Let's look into the memory architecture of the 99/4A

This block diagram came from TI and may be of general interest- a picture is worth a thousand words!

TI-99/4(A) MEMORY ARCHITECTURE

from:
Rockey Mountain
99ers "TIC TALK"



HEAVY LINES INDICATE FEATURES INCLUDED WITH CONSOLE

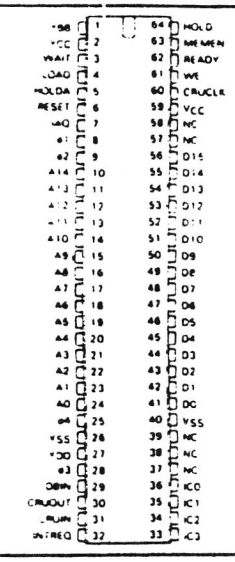
TMS 9900

16-BIT MICROPROCESSOR

FEATURES:

- 16-Bit Instruction Word.
- Full Minicomputer Instruction Set Capability including Multiply and Divide.
- Up to 65,536 Bytes of Memory.
- 3 MHz Speed (4 MHz option).
- Advanced Memory-to-Memory Architecture.
- Separate Memory, I/O, and Interrupt-Bus Structures.
- 16 General Registers.
- 16 Prioritized Interrupts.
- Programmed and DMA I/O Capability.
- N-Channel Silicon-Gate Technology.

TMS 9900 PIN ASSIGNMENTS



900 mil
64 PIN

DESCRIPTION:

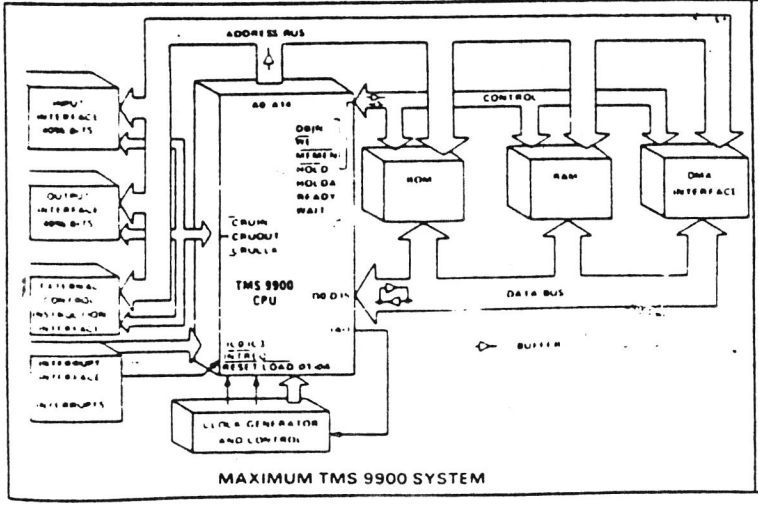
The TMS 9900 microprocessor is a single-chip 16-bit central processing unit (CPU) produced using N-channel silicon-gate MOS technology. The instruction set of the TMS 9900 includes the capabilities offered by full minicomputers. The unique memory-to-memory architecture features multiple register files, resident in memory, which allow faster response to interrupts and increased programming flexibility. The separate bus structure simplifies the system design effort. Texas Instruments provides a compatible set of MOS and TTL memory and logic function circuits to be used with a TMS 9900 system. The system is fully supported by software and a complete series of development systems.

The memory word of the TMS 9900 is 16 bits long. Each word is also defined as 2 bytes of 8 bits. The instruction set of the TMS 9900 allows both word and byte operands. Thus, all memory locations are on even address boundaries and byte instructions can address either the even or odd byte The memory space is 65,536 bytes or 32,768 words.

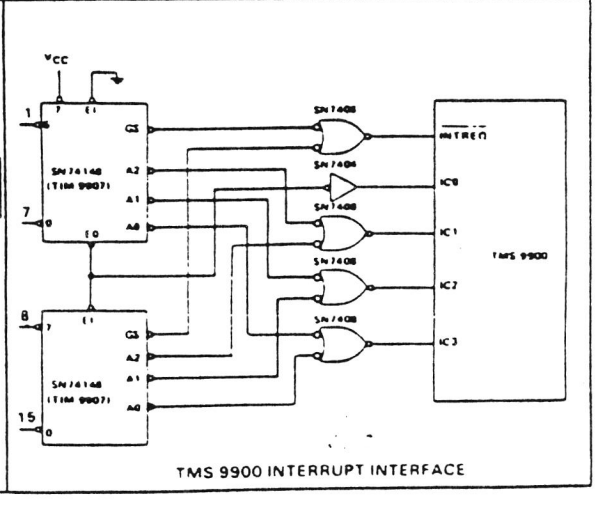
The TMS 9900 utilizes a versatile direct command-driven I/O interface designated at the communications-register unit (CRU). The CRU provides up to 4096 directly addressable input bits and 4096 directly addressable output bits. Both input and output bits can be addressed individually or in fields from 1 to 16 bits. The TMS 9900 employs three dedicated I/O pins (CRUIN, CRUOUT, and CRUCLK) and 12 bits (A3 through A14) of the address bus to interface with the CRU system. The processor instructions that drive the CRU interface can set, reset, or test any bit in the CRU array or move between memory and CRU data fields.

RECOMMENDED OPERATING CONDITIONS

		MIN	NOM	MAX	UNIT
Supply voltage V_{BB}		-5.25	-5	-4.75	V
Supply voltage V_{CC}		4.75	5	5.25	V
Supply voltage, V_{DD}		11.4	12	12.6	V
Supply voltage V_{SS}			0		V
High-level input voltage, V_{IH} (all inputs except clocks)		2.2	2.4	V_{CC} +1	V
High-level clock input voltage, $V_{IH(C)}$		V_{DD}-2		V_{DD}	V
Low-level input voltage V_{IL} (all inputs except clocks)		-1	0.4	0.8	V
Low-level clock input voltage, $V_{IL(C)}$		-0.3	0.3	0.6	V
Operating free-air temperature T_A	NL, JDL	0		70	°C
	JDE*	-40		85	°C



MAXIMUM TMS 9900 SYSTEM



TMS 9900 INTERRUPT INTERFACE

ORDERING INFORMATION:

TMS 9900 NL – 64-PIN PLASTIC DIL (3 MHz) (0/70°C)
TMS 9900 JDL – 64-PIN CERAMIC DIL (3 MHz) (0/70°C)
TMS 9900 JDE – 64 PIN, –40/+85°C
TMS 9900-40 NL – 64-PIN PLASTIC DIL (4 MHz) (0/70°C)
TMS 9900-40 JDL – 64-PIN CERAMIC DIL (4 MHz) (0/70°C)

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

SECTOR 0

Hex	Dec	Meaning (of the 256 bytes in the sector)
00-09	0- 9	The disk name you assigned
0A-0B	10- 11	Number of sectors initialized (ex >0168 = 360)
0C	12	Number of sectors per track (ex >09 = 9)
0D-0F	13- 15	TI identifier - "DSK" or >44534B
10	16	Copy protection (ex >20 = none, >50 = protected)
11	17	Number of tracks (ex >28 = 40)
12	18	Number of sides (ex >01 = single, >02 = double)
13	19	Disk density (ex >01 = single, >02 = double)
14-37	20- 55	not used
38-64	56-100	} This is a bit map of all the sectors on the disk
66-92	102-146	} Use depends on if the disk is SS, DS, SD, or DD
94-C0	148-192	} 1) Take each byte (45 bytes for 360 sectors)
C2-EE	194-238	} 2) Convert to bits (8 bits per byte) 3) Reverse the order of the 8 bits 4) If the bit is "0" then the corresponding sector (0 to 359) is free. If the bit is "1" then the sector is used.
65,93	101,147	}
C1,EF	193,239	} not used
F0-FF	240-255	}

SECTOR 1

Hex	Dec	Meaning
00-01	0- 1	Tells sector of 1st "alphabetic" file directory
02-03	2- 3	Tells sector of 2nd "alphabetic" file directory
:	:	("alphabetic" means that if the filenames were
:	:	sorted this would be the 1st, 2nd, etc filename)
FC-FD	252-253	Tells sector of the 127th "alphabetic" file dir.
FE-FF	254-255	0000 is always after the last filename (if there was only one file then 0000 would be at >02-03)

SECTORS 2-33

Hex	Dec	Meaning
00-09	0- 9	The file name that you used
0A-0B	10- 11	not used
0C	12	File Type bit 0 - 0=fixed 1=variable length bit 4 - 0=none 1=write protected bit 6 - 0=display 1=internal format bit 7 - 0=data 1=program file
0D	13	Number of records per sector (n/a for program)
0E-0F	14- 15	Number of sectors per file
10	16	End of file offset in last sector (n/a for fixed)
11	17	Record size (n/a for program)
12-13	18- 19	Number of records per file (n/a for program) note - the bytes are reversed (ex >0102 = >0201)
14-1B	20- 25	not used
1C	26	Sector where file is located } repeats as needed
1D-1E	27- 28	Number of sectors following } to use any sector note - the bytes are flipped (ex >12 = >2001)

```
**********************************************************************
*                          A  SHORT  PROGRAM                         *
**********************************************************************
```

Time to do a little assembly language programming. Listed below is the sort routine that I gave in the August issue.

Memory	Label	Op-Code	Operand(s)	Comment (equivilant in basic)
7D0C		CLR	R0	This area sets up the program
7D0E		LI	R1,>2	to receive the parameters that
7D12		BLWP	@NR	are passed from the Basic line
7D16		BLWP	@FP	that calls the assembler sort
7D1A		DATA	FI	routine. It's called by -
7D1C		MOV	@FA,R2	
7D20		DEC	R1	CALL LINK("SORT",A(),B)
7D22		MOV	R2,R3	1000 C=INT(B*.75)
7D24		SLA	R3,>2	
7D26		S	R2,R3	
7D28		SRL	R3,>2	
7D2A	L1	MOV	R1,R4	1001 FOR D=1 TO B-C
7D2C		MOV	R2,R5	
7D2E		S	R3,R5	
7D30	L2	MOV	R4,R6	1002 E=D
7D32	L3	MOV	R6,R0	1003 IF A(E)<=A(C+E) THEN 1009
7D34		BLWP	@NR	
7D38		LI	R7,FA	
7D3C		LI	R8,AR	
7D40		MOV	*7+,*8+	
7D42		MOV	*7+,*8+	
7D44		MOV	*7+,*8+	
7D46		MOV	*7,*8	
7D48		A	R3,R0	
7D4A		BLWP	@NR	
7D4E		BLWP	@FP	
7D52		DATA	FC	
7D54		MOVB	@ST,R0	
7D58		ANDI	R0,>4000	
7D5C		JEQ	L9	
7D5E		MOV	R6,R0	1005 A(E)=A(C+E)
7D60		BLWP	@NA	
7D64		LI	R7,AR	1004 F=A(E)
7D68		LI	R8,FA	
7D6C		MOV	*7+,*8+	
7D6E		MOV	*7+,*8+	
7D70		MOV	*7+,*8+	
7D72		MOV	*7,*8	
7D74		A	R3,R0	1006 A(C+E)=F
7D76		BLWP	@NA	
7D7A		S	R3,R6	1007 E=E-C

```
**********************************************************************
*                          A  SHORT  PROGRAM - cont.                 *
**********************************************************************

Memory  Label  Op-Code  Operand(s)   Comment (equivilant in basic)
======  =====  =======  ==========   ============================
 7D7C          JGT      L3           1008 IF E>0 THEN 1003

 7D7E   L9     INC      R4           1009 NEXT D
 7D80          DEC      R5
 7D82          JGT      L2

 7D84          SRL      R3,>1        1010 C=INT(C/2)

 7D86          JGT      L1           1011 IF C THEN 1001

 7D88          MOVB     R3,@ST       1012 RETURN
 7D8C          B        *R11

 7D8E   NR     EQU      >6044        Note - this area defines all
 7D8E   NA     EQU      >6040        of the labels that are used
 7D8E   FP     EQU      >601C        in the program (note that the
 7D8E   FI     EQU      >1200        first five would need differ-
 7D8E   FC     EQU      >0A00        ant addresses if you were to
 7D8E   FA     EQU      >834A        use the Editor Assembler or
 7D8E   AR     EQU      >835C        Extended Basic modules to run
 7D8E   ST     EQU      >837C        this program).

 7D8E          AORG     >7FE8        Note - this area defines where
 7FE8          TEXT     'SORT'       the sorting program is located
 7FEE          DATA     >7D0C        in the Mini Memory module.
 7FF0          END                   Change for EA or EB modules.
```

Here is how the workspace registers are used for the program -

```
R0 = subscript & work area          R5  = B-C
R1 = paramenter & constant 1        R6  = E
R2 = B                              R7  = indirect from address
R3 = C                              R8  = indirect to address
R4 = D                              R11 = GPL return address
```

The program is meant to be run in the Mini Memory module. It may be assembled at any address after >7D0C (be sure and change the ref/def table). If you want to use it with the Editor Assembler or Extended Basic modules, you will need the 32K expansion (the program will go in the 8K region) and the addresses for five of the subprograms (look in the Editor Assembler manual for these).

The advantages of the assembly program are two fold. First is the increase in speed it offers over basic (4.2 vs 22.8 sec/100 #) and this can be speeded up 15 times by using the 32K expansion, since you can directly access the array in the 24K region and don't have to make time calling accesses to the VDP RAM.

The second advantage is in the amount of space each program takes. The basic version eats up 179 bytes while the assembly version needs only 130 bytes. The basic variables use 75 bytes and the assembly a mere 18. Overall count - basic @ 254 bytes and the assembly @ 148 bytes (only 58% of the basic)!

* AN INPUT ROUTINE IN ASSEMBLY LANGUAGE by JIM RICE

To be able to use the assembly routine I've listed in this article, you must have the console, the Editor/Assembler cartridge, a disk drive, and the memory expansion card or peripheral.
Type the program I've listed at the end of this tutorial through the editor. Save the file as "DSK1.INPUTS", then assemble it. The object code file is "DSK1.INPUT". When you assemble it, "R" is the only option you need to use.
This INPUT routine is extremely useful as a subprogram. I have left the listing as simple as possible to make it easily adaptable to any type of application. The routine I've listed here places a nonflashing cursor in the upper left hand corner of the screen. It waits for you to type something. After you do, press enter. It will read what you typed in off of the screen, save it into the CPU RAM scratchpad, and print it back up on the screen in a different location. This saves what you typed in at CPU RAM address(>8300), the starting address of the CPU RAM scratchpad, for easy access for the remainder of the program. This is handy for when you want variables in your assembly programs or for writing to files through the DSR(device service routines)such as disk files, a printer, or a modem.

HERE IS THE LISTING:

```
REF KSCAN,VSBW,VMBW,VMBR  * References to the different memory
* resident routines used
DEF RUN  * Program name defined
CHAR   DATA >007C,>7C7C,>7C7C,>7C7C * Cursor character data.
CURSOR BYTE >A0  * ASCII code for cursor.
SPACE  BYTE >A8  * ASCII code for space character.
RUN
LI R0,>D00  *
LI R1,CHAR   **** Loads the cursor character data into the cursor
LI R2,8      **** ASCII code.
BLWP @VMBW  *
CLR R0
CLR R1
LI R2,>0D00  * Loads register(R2) with ASCII code for "enter" key.
LI R3,>2000  * Loads R3 with byte used to check if key has been pressed
LI R5,>0800  * Loads R5 with ASCII code for backspace(fctn-s) character.
CLR R6
MOVB R0,@>8374 * clears @>8374(tells computer to scan whole keyboard.)
LI R0,1      * loads R0 with 1(tells computer cursor to go in upper left)
More.(A=Abort, any other key to cont.)
J1     LI R1,CURSOR  **** Prints cursor to screen at location indicated by R1
BLWP @VSBW   *
CLR R1
BLWP @KSCAN  * Scans keyboard
MOVB @>837C,R6 * Loads byte that shows whether a key was pressed into R6
COC R3,R6    * Checks if key was pressed.
JNE J1       * If not, goes to J1
MOVB @>8375,R1 * Loads key pressed into R1 from address key code stored
CB R1,R2     * Checks if key pressed is "enter"
JEQ J3       * If so, go to J3
CB R1,R5     * Checks if key pressed is backspace(fctn-s)
JNE J2       * If not, go to J2
MOVB @SPACE,R1  * load space ASCII code to R1
BLWP @VSBW   * Put space where cursor was
DEC R0       * Move cursor back 1 space
JMP J1       * Go to J1
J2     BLWP @VSBW   * Write key detected to the screen.
INC R0       * Move cursor forward 1 space
JMP J1       * Go to J1
J3     MOVB @SPACE,R1 * Load R1 with ASCII code for space
BLWP @VSBW   * Write space to screen where cursor was
```

```
CLR R0
CLR R1
CLR R2
LI R0,1       * Load R0 with starting address of variable inputed.
LI R1,>8300   * Load R1 with CPU RAM SCRATCHPAD startin address
LI R2,80      * Move 80 bytes of characters typed on to screen
BLWP @VMBR    * Read variable input from screen & write it to CPU(>8300)
CLR R0
CLR R1
CLR R2
LI R0,513 * Load R0 with 513(new address on screen to print variable to.)
LI R1,>8300  * Load R1 with address of data to be read(CPU SCRATCHPAD.)
LI R2,80  * Move 80 bytes from CPU scratchpad to new screen location
BLWP @VMBW * Write variable inputed back to new screen location.
LIMI 2  * Enable quit key
JUMP   JMP JUMP  * Wait for quit key to be pressed.
END
```

The actual storing of the variable in CPU RAM stops after the BLWP @VMBR statement. After that, the program is just reading the variable that was typed in back to the workspace registers and printing it back on to the screen at screen location 513. You can change where it is reprinted on the screen by changing the 513 to another number. The number must be between 0 and 768. You can compute the number by multiplying the row you want times 32 and addinG the column to the product.

You don't have to save your variable to the CPU RAM scratchpad. You could theoretically save it to any CPU memory location. If you don't use the CPU scratchpad, I recommend the memory expansion high RAM(>A000->FFE0). To change location the variable is stored at, change the two times >8300 is used in the program to the new address you want to use. Also, you can save memory by making your variable buffers smaller. Mine, in this case is 80 bytes. That means that 80 characters of what is typed in are saved in the CPU RAM scratchpad. You can make your buffer larger or smaller to fit your needs by changing the two 80's located just before the BLWP @VMBR and the last BLWP @VMBW to however many bytes you want stored in your buffer.

To run the routine, the file name is"DSK1.INPUT" and the program name is "RUN" HAPPY COMPUTING!!!!

Jim Rice

End of file

TI-99/4(A) MEMORY ARCHITECTURE

DESCRETE DEVICE	ADDRESS	USAGE
TMS9900 CPU	>0000	CONSOLE ROM (8K BYTE)
	>2000	MEMORY EXPANSION (8K BYTE)
	>4000	DEVICE SERVICE ROMS (8K BYTE)
	>6000	COMMAND MODULE ROM/RAM (8K BYTE)
	>8000	MAPPED PORTS (SEE BELOW)
	>A000	MEMORY EXPANSION (24K BYTE)
MAPPED PORTS		
FAST RAM	>8000	256 BYTES
TMS9919 SOUND	>8400	WRITE DATA
TMS9918A SCREEN	>8800	READ DATA
	>8802	READ STATUS
	>8C00	WRITE DATA
	>8C02	WRITE ADDRESS
TMS5200 SPEECH	>9000	READ DATA
	>9400	WRITE DATA
GROM CONTROL	>9800	GROM READ DATA
	>9802	GROM READ ADDRESS
	>9C00	GROM WRITE DATA
	>9C02	GROM WRITE ADDRESS

As you can see from the above memory table, the amount of usable memory in the BASIC's is limited to the 8K block at >2000 and the 24K block at >A000, for 32K total RAM. The rest of the memory (32K) is considered to be system overhead. This means that this is the needed amount of memory to carry on the required functions of the computer. The addresses given above are known as base addresses. These are the beginning of a block or segment of memory which contains a group of functions, such as the 8K block between >9800 and >9C02.

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

from:
NORTHWEST OHIO
User's Group

USING SPRITES IN ASSEMBLY LANGUAGE

by Dale Wilson

The following is a sprite example for the Mini Memory Module. It may be necessary to make a few minor changes for it to run in the Editor Assembler. To make sprites in assembly language is not too hard but it does require some assembly language skill. All you have to do is load the desired color [in this case for the space character, 32], load in the desired character pattern, and load in a sprite attribute list.

The sprite attribute list contains the dot-row and dot-column position on the screen and also the character number of the desired pattern.

If you want the sprite to have motion, then you also have to set up the motion table. The motion table contains the row and column velocities and two more bytes used by the interrupt routine. The velocities range from >00 (still) to >7F being the greatest positive velocity to >80 being the fastest negative velocity to >FF being the slowest negative velocity. In addition, you must specify the number of sprites that can be in motion in CPU RAM location >837A.

If you are not sure about something in the program, try changing some of the numbers in the data statements and see what happens. It is probably the best way to learn.

```
7D00                    AORG >7D00
                        ****************************
                        *                          *
                        * ASSEMBLY SPRITE GRAPHICS *
                        *     By Dale Wilson       *
                        *                          *
                        ****************************
                        *
                        * SET UP DATA FOR GRAPHICS
                        *
7D00 FF00  CR      DATA >FF00                    COLOR CHAR SET 5
7D02 7001  SD      DATA >7001,>8006,>D000        SPRITE ATTRIBUTES
7D04 8006
7D06 D000
7D08 0505  SS      DATA >0505,>0000              SPRITE MOTION DATA
7D0A 0000
7D0C FF99  PT      DATA >FF99,>99FF,>1824,>42C3 SPRITE PATTERN DATA
7D0E 99FF
7D10 1824
7D12 42C3
                        *
                        * START MAIN PROGRAM SEGMENT
                        *
7D14 0200  GO      LI   R0,>0384      POINT TO VDP COLOR TABLE/SET 5
7D16 0384
7D18 D060          MOVB @CR,R1        GET COLOR DATA IN MSB R1
7D1A 7D00
7D1C 0420          BLWP @>6024        MOVE @CR TO VDP >0384
7D1E 6024
7D20 0200          LI   R0,>0400      VDP ADDR OF SPRITE PATTERN TABLE
7D22 0400
7D24 0201          LI   R1,PT         CPU ADDR OF PATTERN DESCRIPTOR
7D26 7D0C
7D28 0202          LI   R2,8          8 BYTES TO MOVE
7D2A 0008
7D2C 0420          BLWP @>6028        MOVE PT DATA TO VDP RAM
7D2E 6028
7D30 0200          LI   R0,>0300      SPRITE ATTRIBUTE ADDR IN VDP RAM
7D32 0300
7D34 0201          LI   R1,SD         SPRITE ATTRIBUTES TO BE WRITTEN
7D36 7D02
7D38 0202          LI   R2,5          NO. OF BYTES TO WRITE
7D3A 0005
7D3C 0420          BLWP @>6028        WRITE THE BYTES
7D3E 6028
7D40 0201          LI   R1,1          1 SPRITE
7D42 0001
7D44 0A81          SLA  R1,8          PUT 1 IN MSB
7D46 D801          MOVB R1,@>837A     NO. SPRITES WHICH CAN BE IN MOTION
7D48 837A
7D4A 0200          LI   R0,>0780      SPRITE MOTION TABLE VDP ADDRESS
7D4C 0780
7D4E 0201          LI   R1,SS         SPRITE MOTION DATA TO BE WRITTEN
7D50 7D08
7D52 0202          LI   R2,4          NO. OF BYTES TO WRITE
7D54 0004
7D56 0420          BLWP @>6028        PUT MOTION DATA IN VDP RAM
7D58 6028
7D5A 0300  LP      LIMI 2             ENABLE INTERRUPTS TO MOVE SPRITES
7D5C 0002
7D5E 10FD          JMP  LP            WAIT FOR FCTN QUIT
                        *
                        * PLACEMENT OF PGM NAME IN DEF TABLE
                        *
7FF8                    AORG >7FF8
7FF8   53               TEXT 'SPRITE'  PROGRAM NAME
7FFE 7D14               DATA GO        START AT "GO"
701E                    AORG >701E
701E 7FF8               DATA >7FF8     START DEF TABLE AT 7FF8
                        END
```

TI Home Computer Languages

Using a programming language is nothing more than communicating in a way that both you and your computer can understand. There are several different languages that you can use with the 4A. I'll highlight a different language every month and talk about its advantages and disadvantages. Last month, I discussed machine language. This month, we'll look at a higher language: TMS9900 Assembly.

TMS9900 Assembly Language

As I discussed last month, machine language is extremely difficult to use. This difficulty led to the use of a language which lets the programmer use mnemonic (assisting or intended to assist memory) code. This language assigns combinations of letters to represent operations previously only expressed by using binary (0 and 1) numbers. This (somewhat) easier language is called assembly language because the mnemonic commands must be translated (or assembled) back to their binary equivalents in order for the computer to carry them out. The following is a listing of a assembly language program that plays six chimes over and over.

```
        REF  VMBW
        DEF  CHIME
BUFFER  EQU  >1000
H01     BYTE >01
        EVEN
CHIME
        LI   R0,BUFFER
        LI   R1,CDATA
        LI   R2,118
        BLWP !VMBW
LOOP
        LIMI 0
        LI   R10,BUFFER
        MOV  R10,!>83CC
        SOCB !H01,!>83FB
        MOVB !H01,!>83CE
        LIMI 2
LOOP2
        MOVB !>83CE,!>83CE
        JEQ  LOOP
        JMP  LOOP2
CDATA   BYTE >05,>9F,>BF,>DF,>FF,>E3,1
        BYTE >09,>BE,>01,>A4,>02,>C5,>01,>90,>B6,>D3,6
        BYTE >03,>91,>B7,>D4,5
        BYTE >03,>92,>B8,>D5,4
        BYTE >05,>A7,>04,>93,>B0,>D6,5
        BYTE >03,>94,>B1,>D7,6
        BYTE >03,>95,>B2,>D8,7
        BYTE >05,>CA,>02,>96,>B3,>D0,6
        BYTE >03,>97,>B4,>D1,5
        BYTE >03,>98,>B5,>D2,4
        BYTE >05,>85,>03,>90,>B6,>D3,5
        BYTE >03,>91,>B7,>D4,6
        BYTE >03,>92,>B8,>D5,7
        BYTE >05,>A4,>02,>93,>B0,>D6,6
        BYTE >03,>94,>B1,>D8,4
        BYTE >03,>95,>B2,>D8,4
        BYTE >05,>C5,>01,>96,>B3,>D0,5
        BYTE >03,>97,>B4,>D1,6
        BYTE >03,>98,>B5,>D2,7
        BYTE >03,>9F,>BF,>DF,0
        END
```

Assembly Language Program Listing

As you can see, this is a lot of programming just to get your computer to chime for you.

Assembly language allows you strict control over all functions of your computer--you are telling the computer *exactly* what to do.

Assembly language is also useful when tight control must be maintained over the use of memory. Assembly code doesn't take up a lot of memory. You can have an extremely complex program in the built-in memory of your computer.

The real disadvantage of assembly language--and it is not trivial--is that you need skill and a lot of time to program in it proficiently.

However, assembly language is ideal for short, frequently executed program segments. After you write and assemble your code, it can be called from Extended BASIC.

On the 4A there are three pieces of "hardware" that allow assembly language programming: the Editor/Assembler module, the Mini Memory module, or the p-System card (or standalone).

Next month, I'll cover the most common and simple programming language on the 4A: TI BASIC.

Lee Mathews

ASSEMBLY LANGUAGE FROM EXTENDED BASIC

We have an Assembly Language group up and running, and I hope that this group will be able to develop information and programs that will benefit all of our members. In the meantime for those of you who like to program, here are some notes on memory and "PEEK" and "LOAD" addresses that can be used from either X-Basic or "Mini-Memory".

First a little about the TI 99/4A memory. In the console are two memories one is the "CPU", or Central Processing Unit, the other is the "VDP", or Video Display Processor. The VDP contains 1) the screen-image table (containing the actual display on the screen), 2)the pattern generator table (containing the pattern of each character), 3)the character color table (containing the foreground and background color of each character set), 4)the Sprite attribute list (containing all Sprite values), 5) 12K of free memory (RAM, for basic programming).

The 16,383 bytes contained in the VDP RAM are not directly addressable by the CPU. they are memory mapped (more on this in a later article). For this reason you cannot use Call Peek to look into VDP RAM. However for those of you who have "Mini-Memory", you can "CALL PEEKV" and "CALL POKEV" to get into the VDP RAM. To show you what this can do, try this little program. Plug either the Mini Memory Module, or the Editor/Assembler Module into the console and select "TI BASIC" and enter this program.

```
100 FOR T=1 TO 16
110 CALL POKEV(784,16+T)
120 NEXT T
```

This will load the color table into VDP RAM address >0010 and causes the background color of the space character on the screen to change color very, very rapidly. Be prepared to put a delay loop between lines 110 and 120 if you want to see the colors instead of a blur. Now you can start to see the speed of the 9900 chip.

Here's another program. This program is thanks to THE CENTRAL IOWA 99/4A USERS GROUP newsletter "4A FORUM".

For those of you who have the Mini Memory or Editor Assembler cartridge, here is a little program that allows you to see the screen in Normal Mode, Clear Mode (everything is there but is invisible), Text Mode. (40 characters across), Multicolor Mode (each character is made up of four blocks), and Bit Map Mode (you need the 4/A to see each pixel).

(Cont. Page 15)

No need to know assembler to access these. After you run the program, press N,C,T,M,B (the screen may not be readable, but the keys still work).

```
100 PRINT "PRESS A KEY ===> N,C,T,M,B ":
110 CALL KEY(5,K,S)
120 IF K<>78 THEN 140
130 CALL POKEV(-32768,0)
140 IF K<>67 THEN 160
150 CALL POKEV(-32352,0)
160 IF K<>84 THEN 180
170 CALL POKEV(-32272,0,"",-30945,0)
180 IF K<>77 THEN 200
190 CALL POKEV(-32280,0)
200 IF K<>66 THEN 220
210 CALL POKEV(-32766,0)
220 GOTO 100
```

NOW --- back to the CPU, contained in this memory are 2 - 4K ROM (Read Only Memory) chips that contain the basic interpreter, operating system, and the Device Service Routines (which connect up the P-Box and it's cards), the Video Disolay Processor, the Sound Generator, the Speech Synthesizer and the CPU RAM Scratch Pad (a 256 byte RAM memory sector used by the CPU for almost instantaneous operations. Adding a 32K Memory Expansion Card gives you 65,536 bytes of CPU memory. To look at (PEEK) these memory address you must use either the Mini Memory, the Editor/Assembler, or the Extended Basic Module. In Ex-Basic you must "CALL INIT" then "CALL PEEK(#####,##)".

Address 0 to 32767 are positive numbers the same as the address (0 = 0, 32767 = 32767). The address from 32768 to 65535 are accessed by subtracting 65536 from the address number, resulting in a negative address number (32769 - 65536 = -32768, 45010 - 65536 = -20526, 65535 - 65536 = -1).

The form is "CALL PEEK(ADDRESS,RETURN VARIABLE)

Here are some locations to "PEEK" at, and what they do

(-31880,A) RANDOM NUMBER GENRATOR ("A" returns a random number between 0 and 99. You must use RANDOMIZE, first in a program to get a true random number.
(-31879,T) VIDEO DISPLAY PROCESSOR INTERRUPT TIMER ("T" returns a value that is sequentially generated every sixteenth of a second, from 0 thru 255)
(-31878,S) HIGHEST NUMBERED SPRITE IN MOTION. In version 100 XBASIC, this value is always 28. Program execution using this version can be speeded up by disabling the Sprites above the ones being used. A "CALL LOAD(-31878,A)" to POKE the number of the highest numbered moving sprite into this location disables all Sprites above 'A'. For version 110 this value is updated everytime a Sprite is put into motion.
(-31806,64) DISABLE SPRITE MOTION For 110 ("CALL LOAD(-31806,64)" disables all Sprite motion.
(-31877,C) VDP STATUS REGISTER (If C=2 there was a Sprite Coincidence. Using this instead of the "CALL COINC" will give you faster response to a Sprite Coninicidence and more realistic action on the screen)
(-31808,A,B) DOUBLE RANDOM NUMBER GENERATOR ("A and B" will each return a different random number with values from 0 to 255. A RANDOMIZE statement must be executed first. This can be used to give you a random Sprite motion within the allowable range of -128 to 127 if you use the following:

```
CALL PEEK(-31808,A,B)
CALL MOTION(#1,A-128,B-128)
```

(-31806,16) DISABLE THE QUIT KEY ("FCTN" AND "="), At last, at last, at last, before starting to program use the following:

```
CALL INIT :: CALL LOAD(-31806,16)
```

and no more QUIT key mistakes.
(-31806,32) SOUND GENERATOR STOP DISABLED ("CALL LOAD" this and a "CALL SOUND" statement will not turn off, also your console will lock up on the next sound or noise generated.
(-31806,0) Turns all bits off and gets the QUIT KEY, the Sound Generator, and the Sprite motion back to normal.
(-28672,A) If A returns a 96 then the spech synthesizer is attached. If A returns a 0 no speech synthesizer is attached. Saves having to ask.
(-31888,63,255) Disables the disk drives (load your program first) This "CALL LOAD" will gain you the memory being used by the disk drives (its like being able to CALL FILES(0)).
(-31888,55,215) To get the disk drives back (sometimes it doesn't work, as a last resort type "BYE").
(-31931,0) If you have saved a program in XBasic with the protect option and you cannot list or save it again, use this to unprotect the program.
(-31931,128) "CALL LOAD" this and you will protect it again.("CALL PEEK(-31931,P) will tell you if a program is protected.

If you find this type of information useful and can use it. I'd suggest you obtain a subscription to the SMART PROGRAMMER by:

MILLERS GRAPHICS
1475 W. CYPRESS AVE.
SAN DIMAS, CA. 91773

It costs $12.50 per year. He has promised memory maps, memory dumps, and other very interesting and useful data, the data on disabling the Function Quit is from him.

My thanks to MIKE of the BREVARD USERS GROUP (BUG), of Palm Bay, Florida for several of the memory locations used in this article.

Marshall

```
      DEF M1
* This program will check the
* sprite that is defined with the
* SPRNUM equate and test to see if
* a list of characters are on the
* same spot the sprite is on.
*
* Call with CALL M1(A)
* A=0 if no match     A=1 if match
* code written by Jon Burt 8/12/84

GPLWS  EQU >83E0
VSBR   EQU >2028
NUMASG EQU >2008
STATUS EQU >837C
SAL    EQU >300
FAC    EQU >834A
********************************
SPRNUM EQU 1              THESE
CHAR1  EQU 65             CAN
CHAR2  EQU 66             BE
CHAR3  EQU 67             CHANGED
CHAR4  EQU 68
********************************
       EVEN
OFFSET DATA >60
ZERO   DATA >0000,>0000,>0000,>0000
ONE    DATA >4001,>0000,>0000,>0000

SAV11  BSS  2
MYWS   BSS  >20
X      BSS  1
Y      BSS  1

M1     MOV  R11,@SAV11 HOUSEKEEPING
       LWPI MYWS

       LI   R0,SPRNUM   (POINT TO
                        RIGHT PLACE
       DEC  R0          IN SAL)
       SLA  R0,2        mult BY 4
       AI   R0,SAL
       CLR  R1
       BLWP @VSBR

       SWPB R1
       MOV  R1,R8
       AI   R1,R5
       ANDI R1,>00FF
       SRL  R1,3
       SWPB R1
       MOVB R1,@Y

       LI   R0,SPRNUM   (POINT TO
                        RIGHT PLACE
       DEC  R0          IN SAL)
       SLA  R0,2        mult by 4
       INC  R0

       CLR R1
       BLWP @VSBR

       SWPB R1          (RUNNER
       MOV  R1,R9        CODE)
       AI   R1,4
       SRL  R1,3
       SWPB R1
       MOVB R1,@X

       CLR  R10         (RUNNER
       MOVB @Y,R10       CODE)
       SWPB R10
       SLA  R10,5
       CLR  R4
       MOVB @X,R4
       SWPB R4
       A    R4,R10

       CLR  R1          (GET THE
       MOV  R10,R0      CHARACTER
       BLWP @VSBR       AT THAT
       SWPB R1          SPOT)
       S    @OFFSET,R1

       CI   R1,CHAR1    (TEST
       JEQ  ONEJP       CHARACTER
       CI   R1,CHAR2    AGAINST
       JEQ  ONEJP       SELECTED
       CI   R1,CHAR3    ONES)
       JEQ  ONEJP
       CI   R1,CHAR4
       JEQ  ONEJP
*      add more compares here
       LI   R4,4        (LOAD 0
       LI   R3,ZERO     CHOICE IN
       LI   R2,FAC      FP FORMAT)
LOOP0  MOV  *R3+,*R2+
       DEC  R4
       JNE  LOOP0
       JMP  CONT

ONEJP  LI   R4,4        (LOAD 1
       LI   R3,ONE       CHOICE IN
       LI   R2,FAC      FP FORMAT)
LOOP1  MOV  *R3+,*R2+
       DEC  R4
       JNE  LOOP1

CONT   CLR  R0           (SEND BACK
       LI   R1,1        THE INFO TO
       BLWP @NUMASG     EX BASIC)

       LWPI GPLWS       (ENDING
       MOV  @SAV11,R11   HOUSE-
       CLR  @STATUS      KEEPING)
       RT
       END
```

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 E SEQ. IN DATA BUFF
         LI    6,>1D09
         MOV   6,@>8356        POINT TO DEVICE NAME LENGTH
         BLWP  @>6038          DSRLNK TO WRITE ESC E 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  @>6828          PUT DO INTO DATA BUFFER
        MOV   6,@>8356        POINT TO DEVICE NAME LENGTH
        BLWP  @>6838          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  @>6824          PUT LENGTH OF 2 IN PAB
        LI    0,>1E00
        LI    1,CR
        LI    2,2
        BLWP  @>6828          PUT CR LF INTO DATA BUFFER
        MOV   6,@>8356        POINT TO DEVICE NAME LENGTH
        BLWP  @>6838          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  @>6824          PUT CLOSE OP CODE IN PAB
        MOV   6,@>8356        POINT TO DEVICE NAME LENGTH
        BLWP  @>6838          DSRLNK TO CLOSE PRINTER
        DATA  8
        LI    10,50           DELAY
        DEC   10
        JNE   $-2
        MOVB  @S1,@>8C02      RESTORE SAVED DATA TO GRMWA
        SWPB  @S1
        MOVB  @S1,@>8C02
        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
```

This program fills the screen with the alphabet one letter at a time, and then repeats. There is no provision for returning to basic, so you will have to reset the computer to get the program to stop. One note.... when a program is executed in this manner, you cannot return to basic with a B *R11, or a RT command since the contents of register 11 have been changed. The only way left (and I think the best way) is to branch to location >0070. This will return to the next basic statement to be executed.

```
0001 GPLWS  EQU  >83E0      GPL REGISTERS
0002 VSBW   EQU  >2020      VIDEO WRITE
0003 START  LI   R1,161     CHAR CODE FOR A
0004 A      CLR  R0         FIRST SCREEN POSITION
0005        SWPB R1         CHAR TO LEFT BYTE
0006 B      BLWP @VSBW      WRITE TO SCREEN
0007        INC  R0         NEXT SCREEN POS
0008        CI   R0,769     LAST ONE?
0009        JNE  B          NO, LOOP
0010        SWPB R1         CHAR CODE TO RIGHT BYTE
0011        INC  R1         NEXT LETTER
0012        CI   R1,187     PAST Z?
0013        JNE  A          NO, LOOP
0014        JMP  START      YES, START OVER
0015        AORG GPLWS+22   POINT TO RETURN ADDRESS REGISTER
0016        DATA START      OUR STARTING ADDRESS
0017        END             END OF PROGRAM
```

After you assemble the program (use only the R option), load it under Extended Basic. It should begin to execute as soon as it loads.

As I mentioned before, most of my assembly programs interface with Extended Basic. Last year I needed a program that would allow me to examine and change memory locations while an Extended Basic program was running. The following is an Extended Basic program that allows you to do just that. In addition it has both a hex to decimal and a decimal to hex conversion routine. Here is a list of allowed commands.

A - Alter memory. Allows you to change the contents of the memory location you specify.
D - Decimal to Hex conversion.
H - Hex to Decimal conversion.
M - Memory Dump. Gives the contents of consecutive memory locations in both hex and ascii. Optionally outputs to a printer.
AID (FCTN 7) - Prints a list of allowed commands.

```
100 HEX$="0123456789ABCDEF" :: A1(1)=4096 :: A1(2)=256 :: A1(3)=16 :: A1(4)=1
110 CALL PEEK(8198,A) :: IF A<>170 THEN CALL INIT
120 OPEN #1:"PIO" :: CALL CLEAR :: CALL CHAR(48,"003A444E54644488") :: GOTO 360
130 CALL CLEAR :: PRINT "STARTING ADDRESS(HEX) ? ":"PRESS ENTER TO ABORT.";
140 ACCEPT AT(23,25)BEEP VALIDATE(HEX$) SIZE(4):H$ :: IF H$="" THEN 360
150 PRINT "OUTPUT TO PRINTER (Y OR N) ?"; :: CALL SOUND(150,1400,2)
160 CALL KEY(3,K,S) :: IF S=0 THEN 160 ELSE IF K<>89 AND K<>78 THEN 160
170 IF K=89 THEN PR=1 ELSE PR=0
180 GOSUB 690 :: ARD=DEC
190 CALL CLEAR
200 PRINT "PRESS ANY KEY TO PAUSE.":"PRESS ""BEGIN"" TO ABORT.": :
210 DEC=ADR :: GOSUB 740 :: IF ADR>32767 THEN ADR=ADR-65536
220 CALL HCHAR(24,2,62) :: PRINT H$;" =";  :: IF PR=1 AND PR1=0 THEN PRINT1$=">"&H$&" ="
230 FOR C1=0 TO 5
240 CALL PEEK(ADR+C1,A2(C1))
250 DEC=A2(C1) :: GOSUB 740
260 IF C1/2=INT(C1/2) THEN PRINT " "; :: IF PR=1 THEN PRINT1$=PRINT1$&" "
270 PRINT SEG$(H$,3,2); :: IF PR=1 THEN PRINT1$=PRINT1$&SEG$(H$,3,2)
```

```
280 NEXT C1 :: PRINT " ";
290 FOR C1=0 TO 5 :: IF A2(C1)<32 OR A2(C1)>126 THEN PRINT "."; :: IF PR=1 THEN
PRINT2$=PRINT2$&"."
300 IF A2(C1)<32 OR A2(C1)>126 THEN 320
310 PRINT CHR$(A2(C1)); :: IF PR=1 THEN PRINT2$=PRINT2$&CHR$(A2(C1))
320 NEXT C1 :: PRINT :: IF PR=0 THEN 340
330 IF PR1=0 THEN PR1=1 :: GOTO 340 ELSE PRINT #1:PRINT1$;"  ";PRINT2$ ::
PRINT1$,PRINT2$="" :: PR1=0
340 CALL KEY(0,K,S) :: IF S=0 THEN ADR=ADR+6 :: GOTO 210 ELSE IF K=14 THEN 360
350 CALL KEY(0,K,S) :: IF S<=0 THEN 350 ELSE IF K=14 THEN 360 ELSE ADR=ADR+6 :: GOTO 210
360 PRINT : :"COMMAND ? " :: CALL SOUND(150,1400,2)
370 CALL KEY(0,K,S) :: IF S=0 THEN 370
380 IF K=81 THEN CALL CLEAR :: END
390 IF K=72 THEN 460
400 IF K=68 THEN 520
410 IF K=77 THEN 130
420 IF K=65 THEN 580
430 IF K<>1 THEN 370
440 CALL CLEAR :: PRINT "Q=QUIT":"H=HEX TO DECIMAL CONVERSION":"D=DECMIAL TO HEX
CONVERSION":"M=INSPECT MEMORY":"A=ALTER MEMORY" :: GOTO 360
450 GOTO 370
460 CALL CLEAR
470 PRINT "ENTER HEX NUMBER (0 TO FFFF)":"PRESS ENTER TO ABORT.":"? ";
480 ACCEPT AT(24,3)BEEP VALIDATE(HEX$) SIZE(4):H$ :: IF H$="" THEN 360
490 GOSUB 690 :: PRINT
500 PRINT ">";H$;" =";DEC; :: IF DEC>32767 THEN PRINT " OR ";DEC-65536: :ELSE PRINT : :
510 GOTO 470
520 CALL CLEAR
530 PRINT "ENTER DECIMAL NUMBER":"(-32767 TO 65535)":"ENTER 0 TO ABORT":"? ";
540 ACCEPT AT(24,3)BEEP VALIDATE(NUMERIC) SIZE(6):DEC
550 IF DEC=0 THEN 360
560 IF DEC<-32767 OR DEC>65535 THEN PRINT : :"ERROR !!": : :: GOTO 530
570 T1=DEC :: GOSUB 740 :: PRINT : :T1;"= >";H$: : :: GOTO 530
580 CALL CLEAR
590 PRINT "ENTER MEMORY ADDRESS (HEX)":"PRESS ENTER TO ABORT":"?";
600 ACCEPT AT(24,3)BEEP VALIDATE(HEX$) SIZE(4) :H$ :: IF H$="" THEN 360 :: H1$=H$
610 GOSUB 690 :: IF DEC>32767 THEN DEC=DEC-65536 :: A4=DEC ELSE A4=DEC
620 CALL PEEK(DEC,A3)
630 DEC=A3 :: GOSUB 740
640 PRINT : :"MEMORY ADDRESS >";H1$;" = >";SEG$(H$,3,2)
650 PRINT : :"ENTER NEW VALUE (0 TO FF).":"PRESS ENTER TO ABORT.":"?"; :: ACCEPT
AT(24,3)BEEP VALIDATE(HEX$) SIZE(2):H$ :: IF H$="" THEN 670
660 GOSUB 690 :: CALL LOAD("",A4,DEC)
670 PRINT : : :: GOTO 590
680 END
690 REM ** HEX TO DEC CONV ** ENTER H$=HEX#   EXIT DEC=DEC#
700 C1=4-LEN(H$) :: DEC=0
710 FOR C=1 TO LEN(H$)
720 A=POS(HEX$,SEG$(H$,C,1),1)-1 :: DEC=DEC+A*A1(C+C1)
730 NEXT C :: RETURN
740 REM ** DEC TO HEX CONV ** ENTER DEC=DEC#   EXIT H$=HEX#
750 H$="" :: IF DEC<0 THEN DEC=DEC+65536
760 FOR C=1 TO 4
770 A=INT(DEC/A1(C)) :: DEC=DEC-A*A1(C)
780 H$=H$&SEG$(HEX$,A+1,1)
790 NEXT C :: RETURN
```

This program requires Extended Basic to run, and requires the 32K memory expansion option to use the alter memory function. If your printer uses other than the PIO port, CHANGE LINE 120 OPEN STATEMENT