THE PET™ REVEALED

A NICK HAMPSHIRE PUBLICATION

THE PET™ REVEALED

A NICK HAMPSHIRE PUBLICATION

First Edition October 1979
Second Edition January 1980

Published by: Computabits Ltd, P.O.Box 13, Yeovil, Somerset, England.

FORWARD

This book is a collection of discoveries about the PET, how and why it works, and how to use these facts to write better programs and perform more interesting functions. This is the second edition covering both old and new ROM machines, if in doubt as to which ROMs are in your machine then PEEK(50003), in old ROMs this is 0, in new ROMs 1. Although the majority of these facts have never been oficially published by Commodore I would like to thank Commodore U.K. for their assistance in providing much of the information contained in "PET Revealed". Especially Nick Green and Mark Clark of Commodore who were helpfull in providing that information and also in proof reading the manuscript. I would also like to thank Commodore for their permission to publish the circuit diagrams. The discovery of page zero locations and ROM subroutines is principally the work of Jim Butterfield while the Trace programs are the work of Brett Butler, my thanks to them both. I would also like to thank Mark Witkowski for providing some of the other programs and also proof reading the manuscript. It may interest you to know that this book was typeset using a PET system running Commodore's word processor interfaced to a daisywheel printer

Nick Hampshire.

CONTENTS

PET SYSTEM HARDWARE 1

Any computer system large or small, consists of just four basic elements or building blocks. These are 1) Central Processing Unit, 2) Storage or Memory, 3) Input and, 4)Output.

The Basic Elements.

The Central Processing Unit or CPU as it is commonly known, can loosly be regarded by analogy to a human being as the "brain" of the computer. It is inside the CPU that instructions are processed and the arithmetic done. The functioning of other parts of the computer are also controlled by the CPU. The computer stores the instructions which it has been given and the data on which these instructions operate in memory. This memory can be divided into two general categories, main memory and auxiliary backup storage. All the instructions and data required by the machine to perform its current task are stored in main memory. Auxiliary memory provides a permanent storage for sets of data or instructions which may be required by the computer at a later date. Auxiliary storage in the PET consists either of a cassette deck or a floppy disk unit. In the cassette deck the data and programs are stored on magnetic tape, in the floppy disk drive on a magnetic disk. Using these devices the contents of the auxiliary storage can be brought back into main memory as the need arises.

The input allows one to put instructions or information - data - into the computer's main memory. This is most commonly done through a typewriter like keyboard. However inputs can come from other sources besides a keyboard, it could come from the closing of a switch,from a piece of test equipment or even from another computer. The input is also used when information and instructions are transfered from auxiliary memory to main memory.

The output is used by the computer to display the results of its computation. This can be on one of several devices,a video screen,a printer, or to output

the contents of main memory into auxiliary memory. As with input,the output can be to a single device such as a light, to a piece of equipment which the computer is controlling or to another computer.

These then are the basic elements of any computer system and Fig 1 shows how they are connected together and how they interact with the human user. We will now consider in more detail how these four basic elements are implemented on the PET.

The CPU.

The principle component of the CPU circuitry in the PET is the 6502 microprocessor. The internal functioning of this device will be looked at in more detail in chapter 2. For the moment consider this 40 pin integrated circuit as a "black box", since all we are interested in are the inputs and outputs. These can be divided into four distinct groups, there are eight data lines, 16 address lines, 10 control lines and 3 power supply lines, the remaining three IC pins are not connected and have no function, a block diagram of the 6502 is shown in Fig 2. Each of these groups forms what is known as a "bus" which can be defined as being a set of parallel paths used to transfer binary information between the devices in a system.

The "ADDRESS BUS" is used to carry the address generated by the microprocessor to the address inputs of the memory and input/output (I/O) devices. The address bus on the PET is unidirectional since the 6502 is the only component for all the system, except the video circuitry, capable of generating addresses. Since there are 16 address lines the processor can access, i.e. Read or Write into up to a total of 2^{16} or 65,536 words of memory, I/O registers etc. If you look at the circuit diagram for the CPU section of the PET you will notice that the address lines A0 to A15 are divided into two groups. The bottom twelve lines A0 to A11 go to a unidirectional buffer the purpose of which is to increase the power available on each address line. The top four address lines however go to a demultiplexer, this decodes the binary number present on these four address lines, and gives an output on one of the sixteen output lines corresponding to that number. The function of this is to divide the memory area into sixteen blocks each of 4096 bytes of memory, each of which can be selected by means of one of the output lines from the demultiplexer. Why the designers have done this will become obvious when we look at the memory circuitry.

The "DATA BUS" consists of eight bidirectional data lines. During a "WRITE" operation these lines transfer data from the processor to the memory location selected by the address lines. During a "READ" operation data is

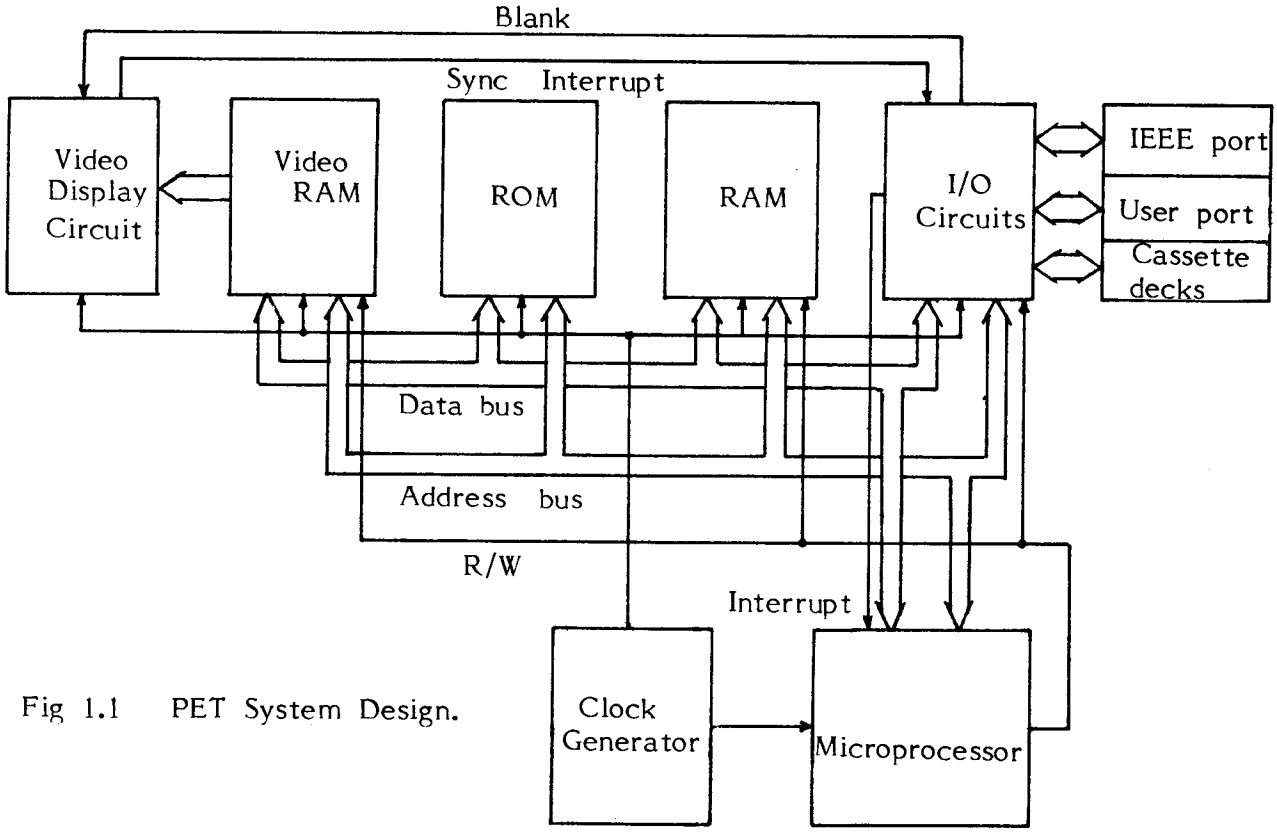


Fig 1.1 PET System Design.

Vss	1	40	$\overline{RES}$
RDY	2	39	ø2 (out)
ø1(out)	3	38	S.O
$\overline{IRQ}$	4	37	ø0 (in)
N.C	5	36	N.C
$\overline{NMI}$	6	35	N.C
SYNC	7	34	R/W
Vcc	8	33	D0
A0	9	32	D1
A1	10	31	D2
A2	11	30	D3
A3	12	29	D4
A4	13	28	D5
A5	14	27	D6
A6	15	26	D7
A7	16	25	A15
A8	17	24	A14
A9	18	23	A13
A10	19	22	A12
A11	20	21	Vss

Fig 1.2 6502 Pinout

transfered from memory to the processor along the same lines. The data bus is thus used to carry all data or instructions to and from the processor,memory, and periferal I/O chips. As with the address lines the data lines have insufficient power on leaving the microprocessor or memory chips to drive the devices to which they are sending data. A bidirectional buffer is therefore used to raise the power levels on the data bus.

To understand the operation of the control lines which comprise the "CONTROL BUS" we must look at each one individualy. On the PET only 5 or 6 of the control lines are used (depending on the model), it will be instructive if we look at all ten since it throws light on some of the limitations of the machine. Since the data bus is bi-directional the processor must have some method of signalling to memory or I/O as to which direction data transfer will take place,i.e. whether memory or I/O is to be "read" or "written" to. This function is performed by the first of our control lines, the R/W or "READ/WRITE" output from the processor. When this line is high (i.e. when the measurable voltage level is greater than 2.4 volts) all data transfers will take place from memory to the processor. If the R/W line is low, there is less than 2.4 volts present, then the processor will write data out to memory.

The processor must not only be able to determine the direction of data transfer, but also the timing of that transfer. It is no use the data arriving at the processor if the processor is not expecting it. Timing is done by the system clock and requires two control lines.o0 is the clock input to the microprocessor from the clock generation circuitry and ϕ2 the clock output to memory etc. Known as a two-phase clock system it consists of two non overlapping square waves, one wave is on the ϕ0 line (the o1 line is identical but not used on the PET)the other wave is on the ϕ2 line. ϕ0 and o1 are known as the PHASE ONE clock pulses and ϕ2 is the PHASE TWO clock pulse, on the PET both these lines have a clock frequency of 1MHz. All the address lines change when there is a positive or high pulse on the phase one line, and data is transfered when there is a positive pulse on the phase two line.

The next group of three control lines are all inputs to the processor and are used to force the processor to perform a program starting at a predetermined location in memory. The first of these is the RST or RESET line which is used to initiate the processor when the machine is first switched on. Obviously when a microprocessor is first switched on the contents of all its internal registers are unknown. There is thus no way that the processor knows which location in memory is the begining of the program (it is assumed that, like the PET operating system and Basic, this program is stored in

read only memory). This then is the function of the reset line and its associated circuitry and software.

The reset circuitry on the PET consists of a 555 timer IC, wired in such a way that when power is first switched on, the reset line is held low for a length of time sufficient to allow the PETs circuitry to come to a fully powered up state. The line then goes high, upon which the processor delays for six clock cycles. It then starts execution of a program whose starting address is stored in memory locations 65,533 and 65,534, these two addresses are known as the reset vector. In machines using the old ROMs this vector is set to hexadecimal FD38 and in new ROM machines to hex FCD1, this is the begining address of the power on reset subroutines.

Whereas the reset line is used to initialise the processor before it starts the execution of a program the two Interrupt lines cause the processor to stop its current program execution and start a new program at a specified location. The two lines are entitled IRQ (Interrupt Request) and NMI (Non Maskeble Interrupt). The NMI line is not implemented on the 8K PET but is available to the user on the memory expansion port of the 16 and 32K machines. The accessability of the NMI line to the user on the dynamic RAM machines is very useful, it allows the user to easily interface circuitry requiring an interrupt.

On the PET the IRQ line is very important since the whole system is designed around the use of interrupts. The scanning of the keyboard, reading and writing to the cassette and internal clock update are all controlled by interrupts. Whenever the interrupt line goes from a high to a low state the processor will finish its current instruction, saving the address of that instruction in an area of memory reserved for such purposes. The processor will then start execution of a program whose starting address is stored in the top two bytes of memory (65,535 and 65,536), this is known as the interrupt vector. The contents of the interrupt vector in machines with the old ROMs is hexadecimal E66B, and in machines with the new ROMs hex E61B. This is the start of the interrupt servicing routine. A separate interrupt vector is used by the NMI, located at 65,531 and 65,532, the contents being hex FEFC.

Interrupts are usually generated by an I/O device as a means of signalling to the processor that there is an input present on that device. Therefore in its simplest form an interrupt servicing routine is a program which reads the input register of the I/O device and stores this value in a specific memory location. Having done this we want the processor to continue the execution of the original program, this is done by having the last instruction in the interrupt servicing routine as a return from interrupt instruction. The only difference between the NMI and IRQ lines is that a programmer can

disable the IRQ line whereas an input on the NMI line will always interrupt the processor.

None of the remaining three control lines, RDY or READY input, an output SYNC and S.O or Set Overflow are used by the PET. When the RDY line is pulled low during a phase one clock cycle it performs the function of halting the processor which will not then execute any instructions until the RDY line goes high. The RDY line, like the NMI line, is available on the memory expansion connector of the dynamic RAM machines, but not on the old 8K machines. A pulse appears on the SYNC output during the phase one of an OP-CODE fetch and stays high for the remainder of the cycle. The SYNC output can be used in conjunction with the RDY input to manually single step the processor instruction by instructioion through a program, a function unfortunately not available on the PET. The S.O input is a means of externally setting the overflow flag in the processor, it is designed to be used by future I/O devices in the 6500 series family of ICs. The power requirements for the 6502 are very simple, the system bus requires just a single 5 volt power supply line and a ground line. In fig 2. VCC is the 5 volt line and VSS is the ground.

Memory

As we have seen the sixteen lines of the address bus allow the processor to access up to 65,536 words or bytes of memory, the basic 8K PET uses 23,576 of these locations. We can divide the memory occupying this space into three types, Random Access Memory or RAM, Read Only Memory or ROM, and I/0 registers. The users programs and data are stored in RAM, this type of memory allows the user to both read data from and write data to a memory location, in the 8K PET there is 8K of RAM (1K is 1024 memory locations).

RAM however has disadvantages, when the power to the machine is turned off the contents of RAM memory is erased. If only RAM memory were used we would not have a computer like the PET, which powers up straight into BASIC when the power is turned on. This requires programs to be permanently stored in the machines memory. ROM performs this function, permanently storing the operating system software, (this includes things like the power-on reset program) and the BASIC interpreter. As its name implies the processor can not write data into ROM memory it can only read the contents of these locations, ROM memory in the PET occupies 14K or 14,336 bytes of memory.

The PET is designed around a system of computer architecture known as "memory mapped I/0", briefly, all input and output from the computer is treated as memory locations. In the PET memory 2048 bytes are dedicated to this purpose and are divided between four I/0 devices,

four bytes each to the two PIAs sixteen bytes to the VIA and the remaining 1024 bytes to the video circuitry, we shall be examining these in detail later.

The designers of the PET have split the total memory area into sixteen blocks, each of 4K bytes. This is done by feeding the four most significant address lines into a demultiplexer, from which each of the sixteen output lines can be used to select a unique 4K memory block. There are several versions of the PET, the principle difference between them, besides changes in the software, is the use of different types of RAM chip. The old 8K machines used4K bit static RAMs, these were one of two types the 6550 and the 2114. Both these chips are functionally indentical in most respects since they are organised as 1K by 4 bits. The latest versions of the static RAM 8K machines used the 6550.

A 4K RAM block in th old machines consists of eight memory chips organised in pairs, where each pair contains 1K by 8 bits of memory. Since 1K (1024) is equal to 2^{10} any memory location within the 1K block can be accessed by using the bottom ten lines of the sixteen line address bus. Each memory chip has a set of inputs known as "chip select inputs" there are four on the 6550, these can be used to selectively turn a particular chip off or on and are thus functionally similar to the address inputs. It is these chip select inputs which are used to turn on a particular 1K pair of memory chips, the location of that 1K being determined by address lines 10 and 11 and one of the sixteen 4K block select lines. Herein lies the reason for the division of memory into 4K blocks, since there are only four chip select lines on the 6550 the processor could only access 2^4 x1K or 16K of memory if we connected these inputs to lines 10, 11, 12 and 13 of the address bus. Obviously this is unsatisfactory, and can be remedied if the memory is divided into 4K blocks each of which is selected by a single line going to one of the chip select inputs on the chips in that block. For the 6550 to be turned on two of the chip select lines must be connected to 5 volts and the other two to 0 volts or ground. With careful wiring, this fact can be used to remove any need for decoding of the two address lines (10 and 11) thereby simplifying the circuit and reducing the number of components.

The new 32K and 16K dynamic RAM machines use the 4116 memory chip and the dynamic 8K the 4108. These two RAM chips are pin compatable, with the 4116 having 16K bits of memory and the 4108 8K bits. This is useful since it allows the same circuit board to be used for all sizes of machine. Memory on the 16 and 32K machines is organised as two banks each of 16K bytes, only one bank being implimented in the 16K. The 4K block select lines are not used in the dynamic machines and are replaced by a bank select circuit controlled by address

lines 14 and 15. The circuit diagrams show the circuits for the dynamic RAM systems implemented as a 32K machine.

The operating system and Basic are stored in ROM, on the old 8K machines in seven 16K bit chips of the 6540 type, in the new dynamic PET, in four chips of the 2332 type. The 6540 ROMs are organised as 2K byte memory blocks thus any address can be accessed using the bottom eleven lines of the address bus. The chip select lines on the 6540 are used to select the 4K block being accessed and to determine which of the two chips in the 4K block is to be read. The inputs to the chip select lines of the 6540 being provided by address line 11 and block select lines 12,13,14, and 15.

In the dynamic RAM PET the 2332 ROMs used are organised as 4K byte blocks with the chip select lines on each ROM being connected to one of the block select lines. Sockets are provided for seven ROMs, though only four are required for the operating system and Basic. Of the extra empty sockets, one in memory area B000 to BFFF hex is required for the Commodore program security ROM. The other two empty sockets are available for user written machine code software which can be programmed onto a 2732 EPROM (this is pin compatable with the 2332 ROM).

The designers of the PET have given the user the capability of expanding the amount of memory, either RAM,ROM or I/O, up to a maximum user memory area of 44K bytes. On all models this extra memory circuitry can be connected to the address, data and control buses of the PET via the memory expansion connector on the side of the machine. In the new dynamic PET, memory can also be expanded by either inserting extra RAM chips into the sockets provided or exchanging the 4108 chips for 4116,this will double memory capacity. As already mentioned ROM memory can be expanded on these machines by utilising the empty ROM sockets.

The number of bytes of user memory available is displayed on the screen when the machine is switched on, this is a fairly good way of detecting any memory faults. If on an 8K machine the number of bytes free is less than 7167 then there is a memory fault in the byte at location - number of bytes free + 1025. Some memory faults are however not detected by the system diagnostics, to find some of these a slightly more sophisticated diagnostic program is required. One way of doing this is to load each byte with - 10101010 - or decimal 85, then test if the byte contains this bit pattern. If it does, then the same byte is loaded with - 01010101 - or decimal 170 and again tested. Other values used to load and test each byte are 0 and 255. This procedure will detect most faults due to pattern sensitivity or leaky bit locations.

The following Basic program will test the memory of

a standard 8K PET, and indicate the location and bit pattern of the fault. It is written in Basic and therefore prevents one from testing the bottom 2K of memory, rewritten in machine code this problem could be overcome. The program also detects time dependent errors by displaying the time taken to test each 1K block. Though this program tests only an 8K machine it could be modified for larger machines. The program starts by requesting the start and end memory locations of the test.

```
5 INPUTA,B
10 PRINT"[CLEAR]":TI$="000000"
20 FORI=ATOB
21 FORY=1TO4
22 READN
23 POKEI,N:X=PEEK(I)
24 IFX=NTHEN26
25 GOSUB200
26 NEXT
27 RESTORE
30 DATA0,85,170,255
110 PRINT"[HOME,DOWN 11]";I-1024,I,TI$
120 NEXT
130 PRINT"END OF TEST"
140 END
200 IFX=1ORX=2ORX=4ORX=8ORX=81ORX=84ORX=87ORX=93THEN300
210 IFX=162ORX=164ORX=171ORX=174ORX=247ORX=251ORX=253ORX=254THEN300
220 IFX=16ORX=32ORX=64ORX=128ORX=21ORX=69ORX=117ORX=213THEN350
230 IFX=42ORX=138ORX=186ORX=234ORX=127ORX=191ORX=223ORX=239THEN350
300 A$="I":GOTO400
350 A$="J"
400 IFI<=2047THEN500
410 IFI<=3071THEN510
420 IFI<=4095THEN520
430 IFI<=5119THEN530
440 IFI<=6143THEN540
450 IFI<=7167THEN550
460 GOTO560
500 B$="2":GOTO600
510 B$="3":GOTO600
520 B$="4":GOTO600
530 B$="5":GOTO600
540 B$="6":GOTO600
550 B$="7":GOTO600
560 B$="8":GOTO600
600 PRINT"YOU HAVE A FAULT AT ADDRESS ";I;"IN ROW ";A$;B$;".",N,X
605 RETURN
```

Input and Output.

The input and output devices on the PET are the keyboard, the two cassette decks (one internal one external), the user port, the IEEE 488 interface and the video display. These devices all have one thing in common, whether they are input or output, they are all located within the addressable space and are thus treated by the operating system software as memory locations. This use by the designers of memory mapped I/O means that we can look at the PET I/O in two ways, first as a standard logic circuit. Second and more interestingly from the PET users point of view we can look at the PET I/O as a memory map, from which we can see the exact function of every bit in every location figure 4 is such a map.

The main I/O of the PET, excluding the video circuitry, is performed by three LSI integrated circuits, they are two 6520 Peripheral Interface Adapters (PIA) and one 6522 Versatile Interface Adapter (VIA). To the processor these chips look like RAM memory located in the upper half of memory block 15 and are selected by address line 11 and select line 15 connected to two of the chip select inputs on each chip, (in the case of the 6522 these lines are combined by an'AND' gate whose output goes to the chip select). Each of the three chips is exclusively accessed by connecting the remaining chip select input to one of the address lines, thus PIA number 1 uses A4, PIA 2 uses A5 and VIA A6.

Within each I/O chip there are a set of register, there are four in a PIA and sixteen in a VIA, these are memory locations accessed by the processor. These registers are addressed by the bottom two address lines in the case of a PIA and the bottom four for a VIA , data enters or leaves via the eight bit data bus. As with RAM memory the data direction on the data bus is controlled by the R/W line and its timing by the o2 clock line. Unlike RAM the I/O chips have a control line output, this is the IRQ line which signals to the processor that an input is present on one of the chips.

The peripheral I/O of all these ICs are identical the difference between them lying in the use of the internal registers and the effect they have on the outputs, these will be looked at in chapters 4 and 5. The output from each chip consists of two eight bit bidirectional I/O ports and four control lines, two to each port. Each line in the eight bit port can be programmed by the user to be either an input or an output, the eight lines could be all inputs, all outputs, or a mixture of both. Of the four control lines on each chip, two function as interrupt inputs and the other two can be either interrupt inputs or peripheral control outputs.

The keyboard is wired as a ten row eight column matrix, when a key is depressed one of the row lines is connected to one of the column lines. The eight column lines which are normally at a high logic load are connected to a peripheral I/O port on 6520 (1) and are configured by the operating system software as inputs. If there was a low voltage on all the row lines, then an input, where an input is a low logic load from a column line to the processor, could come from any one of ten keys on that column line. This is overcome by having one row line "off" at a time, and scanning this line across all ten lines. Only when the row line on which the depressed key lies is "off" will there be an output on one of the column lines. The ten row lines are obtained from the demultiplexed output of four lines on the second I/O port on 6520 (1). Though the keyboard is organised as an eight by ten matrix only 73 of the possible 80 keys are used on the PET. The control, scanning and decoding of the keyboard are all done by a set of subroutines within the operating system software which tests the keyboard about sixty times a second for an input. These subroutines are called by an interrupt, generated by the clock circuitry and input to the processor via the CBI pin on 6520 (1). It is this interrupt software which, besides scanning the keyboard also updates the PETs real time clock and controls the blinking of the cursor. Chapter 3 will deal with this and other operating system software.

The IEEE 488 port uses the second of the 6520 PIA chips to provide the majority of the required I/O lines. One of the eight bit I/O ports on the 6520 is designated by the operating system software as input and the other as output, a bi-directional data buffer is used to connect each input line to its equivalent output line. This creates a true eight line bi-directional data bus and conforms generaly to the IEEE 488 standards. A similar data buffer is used to provide the four bi-directional control lines used by the port, the eight input and output lines supplying this buffer are made up of the three control lines of the 6520 and five lines from one of the I/O ports on the 6522. The remaining three control lines of the IEEE port are not bi-directional in nature and are provided by one of the control inputs on 6520 (2), by an output of the current state of the reset line and by a single line from the second output port of 6520 (1). Two of the control line inputs to the IEEE port function as processor interrupts, these can be used by devices connected to this port to signal to the PET that they are ready to input or accept data. By generating an interrupt the processor can be forced to jump to the relavent subroutines, either user written, or within the operating system which control the functioning of the IEEE 488 port.

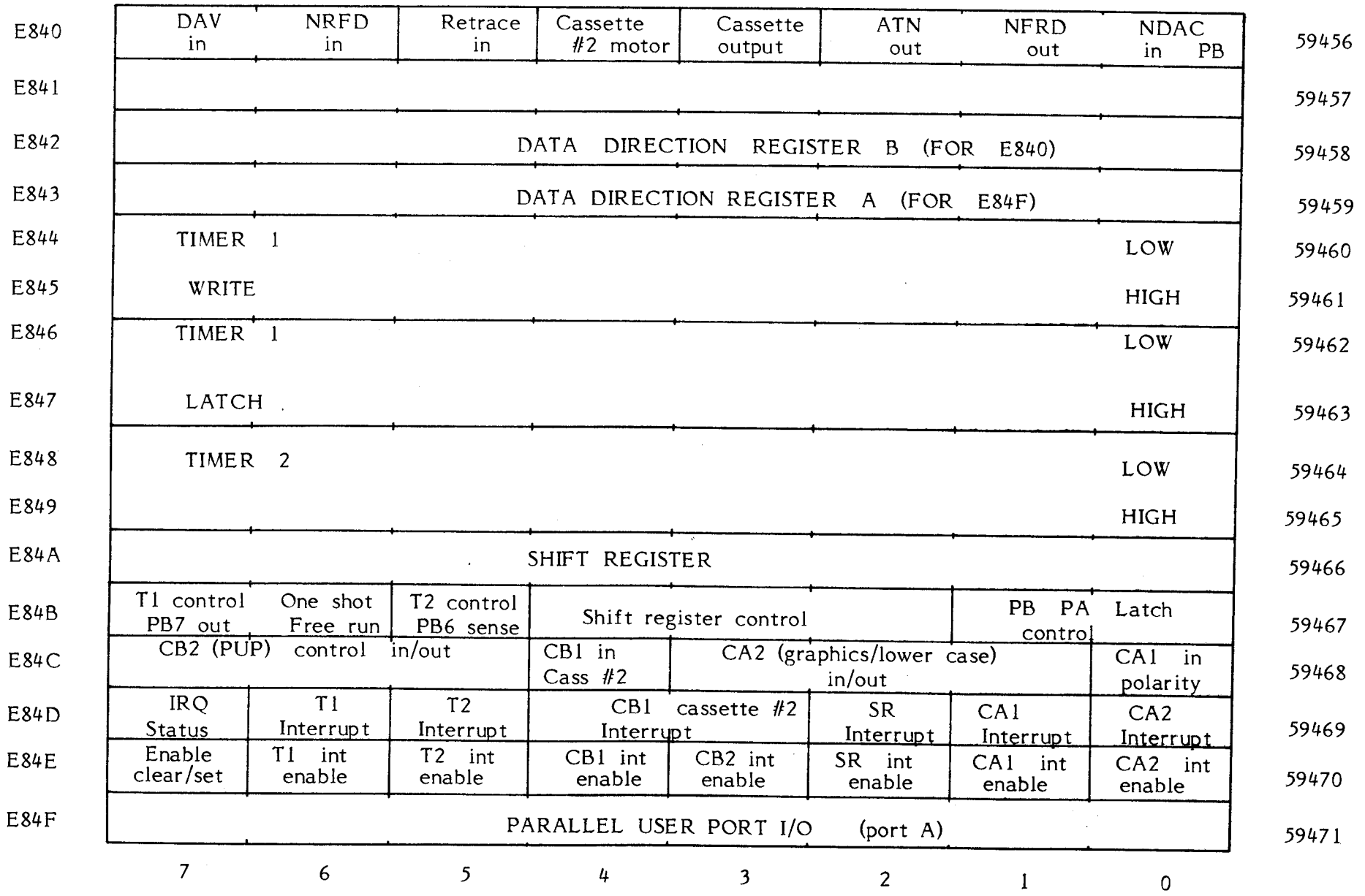
Fig 1.3 SYSTEM I/O MEMORY MAP

PIA 1 (6520)

E810	Diagnostic Sense	IEEE EOI in	Cassette Sense #2 #1	KEYBOARD ROW SELECT	PA	59408
E811	Tape #1 Input flag	....	Screen blank output (old 8K only) IEEE EOI out CA2	DDRA Access	Cassette #1 Read control CA1	59409
E812	KEYBOARD ROW INPUT					59410
E813	Retrace I flag	....	Cassette #1 motor output CB2	DDRB Access	Retrace interrupt Control CB1	59411

PIA 2 (6520)

E820	IEEE INPUT					59424
E821	ATN I flag	....	IEEE NDAC out CA2	DDRA Access	IEEE Control ATN in CA1	59425
E822	IEEE OUTPUT					59426
E823	SRQ I flag	....	IEEE DAV out CB2	DDRB Access	IEEE Control SRQ in CB1	59427


E840
DAV in
NRFD in
Retrace in
Cassette #2 motor
Cassette output
ATN out
NFRD out
NDAC in
PB
59456
E841
59457
E842
DATA DIRECTION REGISTER B (FOR E840)
59458
E843
DATA DIRECTION REGISTER A (FOR E84F)
59459
E844
TIMER 1
LOW
59460
E845
WRITE
HIGH
59461
E846
TIMER 1
LOW
59462
E847
LATCH
HIGH
59463
E848
TIMER 2
LOW
59464
E849
HIGH
59465
E84A
SHIFT REGISTER
59466
E84B
T1 control PB7 out
One shot Free run
T2 control PB6 sense
Shift register control
PB PA control
Latch
59467
E84C
CB2 (PUP) control in/out
CB1 in Cass #2
CA2 (graphics/lower case) in/out
CA1 in polarity
59468
E84D
IRQ Status
T1 Interrupt
T2 Interrupt
CB1 cassette #2 Interrupt
SR Interrupt
CA1 Interrupt
CA2 Interrupt
59469
E84E
Enable clear/set
T1 int enable
T2 int enable
CB1 int enable
CB2 int enable
SR int enable
CA1 int enable
CA2 int enable
59470
E84F
PARALLEL USER PORT I/O (port A)
59471
7
6
5
4
3
2
1
0

VIA (6522)

The user port serves two functions, firstly as a user programmable eight line I/O port with two associated control lines. Secondly as a source of the relavent lines required by the service engineers diagnostic equipment. The programmable I/O and control lines are provided by one half of the 6522 VIA chip. Of the control lines, one is an interrupt input, and the other can be either an interrupt or an output line. When the processor is interrupted by one of these lines it halts, and jumps to a machine code interrupt handling subroutine which has been written by the user. The starting address in the old 8K PET is contained in memory locations 537 and 538 decimal. In the new dynamic PET the locations used are 144 and 145 decimal. Of the diagnostic lines the most interesting to the user are three video output lines which with a bit of simple circuitry allow the screen to be displayed on an external video monitor.

The circuitry which interfaces with the two cassette decks, one internal one external, is identical for each cassette deck. Just four lines are used, three outputs, and one input. The outputs which come from the I/O port lines of the 6522 and 6520 (1) are cassette write, this is common to both decks, motor control and cassette switch. The input from each deck is the cassette read line and these go to the remaining interrupt inputs one on the 6522 and the other on 6520 (1). Thus during a read operation every time a pulse is input from tape via the cassette electronics the processor is interrupted, and the tape read subroutines called. These convert the serial stream of pulses into eight bit words which are then stored in the correct memory location.

The Video Circuitry.

The video display also uses a memory mapped technique, 1K of memory is used from 8000-83E7 hex (32,768-33,768 decimal) where each byte contains the coded representation of a character in a particular position on the screen. There are 25 lines each of 40 characters on the PETs display a thousand characters in all, thus a 1000 memory locations are required by the video circuitry. The processor can write any character to any location on the screen simply by placing the correct byte of data into the correct memory location. This can be simply demonstrated using the POKE command in Basic, POKE 33268,42 will print an asterisk in the middle of the screen.

A unique function of this block of memory is that it is not only accessed by the processor but also by the video circuitry. There are two seperate ten line address buses, one from the processor, and the other from a video address generator crcuit. Normally the memory locations are accessed about sixty times a second. There

are two seperate ten line address buses, one from the processor, and the other from a video address generator circuit. Normally the memory locations are accessed about sixty times a second. There are two address bus inputs and two data buses, one going to the processor and the other to the address inputs of a special ROM chip known as a character generator. Each character is stored in the video RAM as a coded byte of data, the code used is ASCII (American Standard Code for Information Interchange), each letter or number, and in the case of the PET, graphics character, has a unique eight bit code.

The character generator has eleven address lines the upper eight of which are connected to the video RAM data bus, the bottom three to a binary counter, the input to which comes from the video timing circuit. The eight output lines from the character generator are connected to a parallel in/serial out shift register. This converts each byte of data into a stream of pulses, and combined with some timing pulses, provides the inputs to the PET TV monitor. Each character is stored in the character generator as eight bytes of data, this is the reason for the bottom three address lines being connected to a binary counter, and can be thought of as an eight by eight matrix. Each bit in the matrix corresponds to a point on the screen, a pixel, the PET screen is 320 pixels wide and 200 deep. A bit can be either 'on' giving a bright dot on the screen or 'off' leaving a dark space. If you look carefully at the screen you will see that each character is built up from dots organised as eight rows and eight column.

All this requires very accurate and complex timing, the majority of the video circuit is devoted to this purpose. This circuit which is crystal controlled for great accuracy also provides the oO clock line to the processor and the keyboard interrupt. As there are two address, and two data busses, going into and out of the video RAM, some method must be used to avoid conflicts between the processor and the video circuitry. On the address bus a data selector chip is used, this acts like a change over switch and is controlled by a single input line, which is in fact memory block select. If this line is in a 'high' state then the video RAM address lines are connected to the processor address lines, if it is in a low state, then they are connected to the video address generator. A tri state buffer is placed on the data bus between the video RAM and the processor, this acts like a valve opening and connecting the two busses when the processor is accessing the video RAM. The opening and closing of this valve is controlled by the Read/Write line and memory block select line number eight. The random flashes seen when the computer is PEEKing to the video RAM is because in a static RAM machine the data bus is still connected to the character

generator while the processor is accessing memory.

The PET as a system.

The aim of this chapter has been to give an outline of the PETs circuitry, and how the different sections of that circuitry form a complete system. To anyone other than a service engineer an intimate knowledge of the PETs circuitry is interesting but unnecessary.

The reason being that from the users point of view the entire circuitry can be looked at in terms of a memory map. The design of the whole machine relies upon the operating system software, we have seen this in the extensive use of interrupts and the fact that I/O uses memory locations. This means that an ingenious user could change the design of the machine simply by rewriting the operating system software. Armed with such a memory map a PET user can, even from a Basic program, control the machines I/O in an infinite variey of ways, opening up a whole new range of applications.

THE 6502 MICROPROCESSOR 2

When a program is run on the PET all the instructions are performed by one component, the microprocessor. This particular device, there are a range of different microprocessors, is manufactured by MOS Technology and known as the 6502. It is an eight bit microprocessor, eight bits meaning that during each instruction or operation cycle, eight bits of data are operated upon or transferred simultaneously. In Chapter 1, the microprocessor was considered as being just a "black box" with inputs and outputs. However, to use the PET to its maximum potential, a knowledge of the internal functioning of the microprocessor is vital, particularly if the user is writing programs in machine code.

An Overall View

A block diagram of the internal structure (or system architecture as it is called) is shown in figure 2.1. This may appear rather complex, but it can be divided into two sections. One called the control section, the other the register section. The control section lies on the right side of the drawing, the register section on the left. All the processing is carried out within the register section of the chip, instructions obtained from program memory are implemented by a series of data transfers within this section. Each of the 56 different instructions which the 6502 recognises involves a unique set of data transfers. It is the control section which recognises the instruction, and initiates the correct sequence of data transfers. The instructions enter the processor via the data bus and are latched into the instruction register to be decoded by the control logic. Since most instructions require more than one data transfer within the register section, a source of timing signals is required to ensure the correct sequence, this is done by the timing control unit.

Each data transfer which takes place within the register section, is the result of the decoding of the

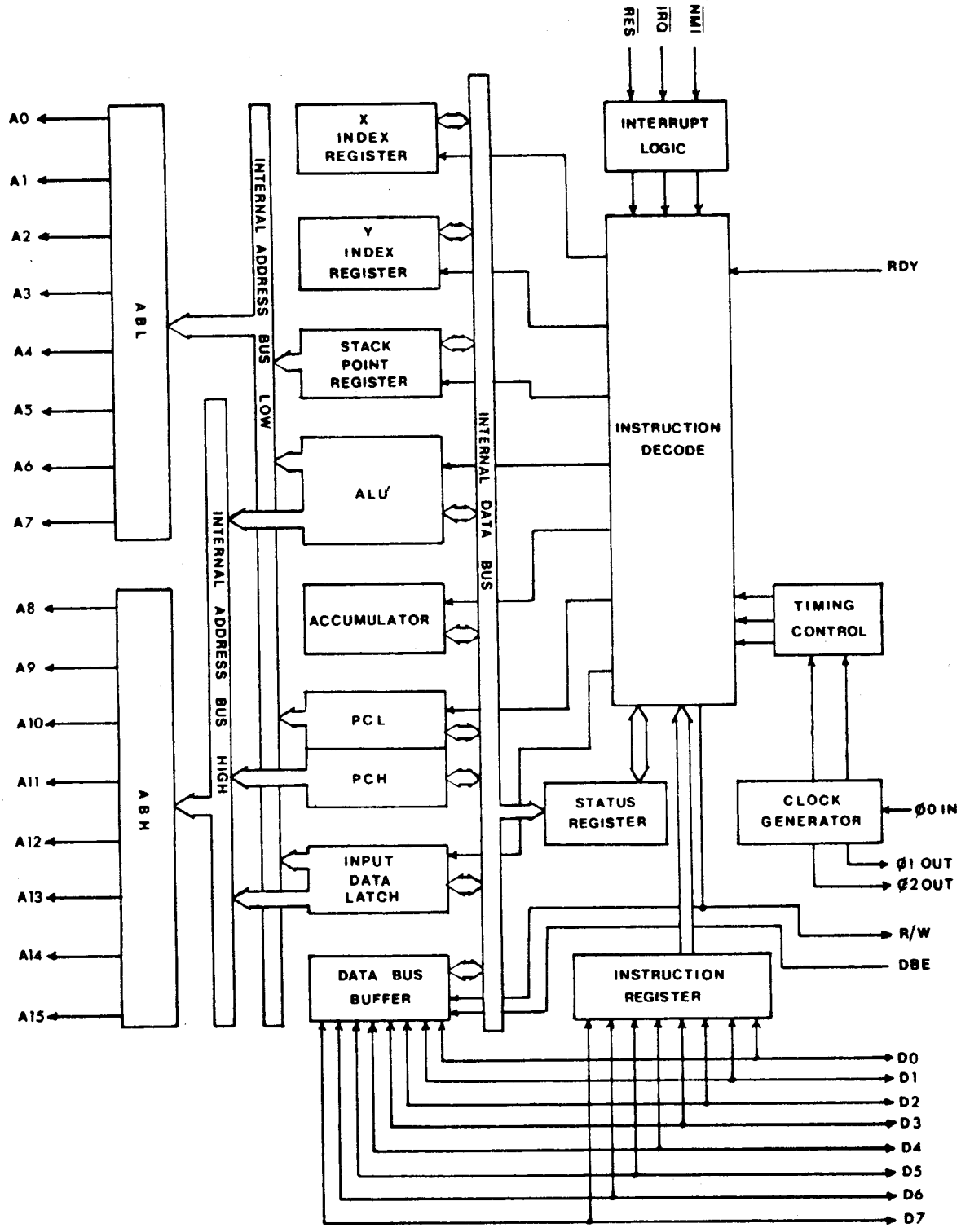


Fig 2.1 6502 Block Diagram

instruction register and the timing control unit by the control logic, whose outputs enable the relevant registers. When programming at a machine level a primary concern is the control and manipulation of data within the processors registers. To understand the function of the microprocessors instruction set, one must understand the function of its registers.

The Accumulator and the Arithmetic Unit

Figure 2.1. shows that the registers communicate with each other via an internal eight line data bus, connected to the computer system data bus by the data bus buffer. One of the simplest types of data transfer is between memory, and an internal register, such as the accumulator.

The accumulator has no exact function, a kind of general purpose register, it is here that data on which operations are being performed is stored. If you want to move a byte of data from one part of memory to another it has to be temporarily stored in the accumulator. Similarly the accumulator is used to store the intermediate and final results of a logic or arithmetical operation.

Data tranfers between the accumulator and memory, which, since the PET is a memory mapped system also includes I/O, are very important and account for about 40% of all the instruction used in a machine code program. To move a byte of data from one memory location to another then two instructions are required:

LDA,M1 - Load accumulator with contents of first memory location
STA,M2 - Store contents of accumulator in second memory location

Memory locations M1 and M2 are accessed by one of a variety of addressing modes, these will be looked at later in the chapter. Having loaded a byte of data into the accumulator the processor can be instructed to perform arithmetic or logical operations upon it. Although these are the kind of functions expected of a computer, only about three percent of all instructions in a program fall within this category. Since the 6502 is an eight bit machine all the arithmetic and logical operations are between two eight bit numbers, the numbers used are limited to a range of between 0 and 255, a limitation which has to be overcome by programming techniques.

The problem of being unable to store a number greater than 255 in the accumulator or memory occures when adding two numbers whose sum is greater than 255. This is overcome by giving the accumulator a nineth bit, called the carry. The carry bit, or flag as it is known,

is one bit in the processor status register, and is set when the contents of the accumulator exceeds 255. All this applies to the performance of binary arithmetic by the processor, the 6502 is fairly unique in that it can also do decimal arithmetic. In this mode each byte contains two binary coded decimal numbers and can have a range from 0 to 99. As in the binary mode when the addition of two numbers gives a result greater than 99, the carry flag is set to indicate the fact. The processor is placed in the decimal mode by a "set decimal mode" instruction, SED, which turns on another bit within the processor status register.

There are two basic arithmetic instructions, ADC - which is add memory to accumulator with carry, and SBC - which is subtract memory from accumulator with borrow. Both instructions can be either binary or decimal in nature and can use a variety of addressing modes to indicate the memory location.

The ADC instruction adds the value of the data in the memory location, plus the carry from the previous operation, to the value in the accumulator, storing the result in the accumulator. If the result exceeds 255 in the binary mode, or 99 in the decimal mode, then the carry flag is set,if the result is zero then the zero flag is set. An example, if we want to add the two numbers, 25 and 189, and store the result in memory location 10 (decimal) we could use the following sequence of instructions:

```
CLC        18          (this clears the carry flag)
LDA 25     A9 19       (Load accumulator with 25)
ADC 189    69 BD       (Add 189 to accumulator and
                        carry)
STA 10     8D 0A 00    (Store result in location 10)
```

The instructions in the left column are in mnemonic code, followed by a decimal number or memory location. The same sequence of instructions appears on the right, written in a numerical form, in this case using hexadecimal notation, showing how instructions and data would be stored in memory. Addition of two numbers with values greater than 255 needs a process known as multiple precision addition, calling for the use of the carry flag. Adding two sixteen bit numbers, requires two additions. The carry is first cleared and the two lowest order bytes, (a sixteen bit number would be stored in two bytes of memory) added together. The result of this addition is stored in a memory location as the low order byte of the result. Now the two high order bytes are added, plus any carry generated by the first addition, the sum stored as the high order byte of the result. Using this method numbers of any size can be added

together, whether the processor is in binary or decimal mode.

Addition can be performed on signed numbers, positive numbers added to negative numbers, or two negative numbers added. The sign is stored as bit seven of the highest order byte, a zero for positive and a one for negative. Addition takes place as in ordinary arithmetic, the only exception being that the carry flag for the highest order byte is replaced by the overflow flag. This performs the same function but records an overflow or carry from bit seven, rather than bit eight. Negative numbers are stored not as ordinary binary numbers but as two's compliment, which is best described as the inverse of that number minus one. All the ones become zeros and vice versa for all bits, except bit one, thus binary five is normally 00000101 - in twos compliment form it becomes: 11111011.

The SBC instruction subtracts the value of data in a memory location, and borrow, from the value in the accumulator, storing the result in the accumulator. Two's compliment arithmetic is used throughout. The borrow flag is the same as the carry flag used in addition, whereas before an addition the carry flag is always cleared, before a subtraction it is always set. The result of subtraction affects the carry or borrow flag, it is set if the result is greater than or equal to zero. Similarly for subtraction of signed numbers the overflow flag is set if the result exceeds +127 or -127 for single precision seven bit arithmetic. The SBC instruction can be used with either binary or decimal numbers with both multiple precision and signed arithmetic. To subtract two decimal numbers, say, 18 from 27 use the following sequence of instructions, the decimal mode is used to illustrate its function:

```
SED        F8          (set decimal mode instruction)
SEC        38          (set borrow flag)
LDA 27     A9 27       (load accumulator with 27)
SBC 18     E9 18       (subtact 18 from accumulator
                        and borrow)
STA 10     8D 0A 00    (store result in location 10)
```

The instructions on the left are in mnemonic code, on the right in hexadecimal, note that in the decimal mode the hexadecimal and decimal numbers are the same.

The 6502 instruction set does not include instructions to perform multiplication or division. Users requiring them must write subroutines to perform these functions, or use the subroutines within PET basic. Multiplication is a process of repeated addition: 3 x 5 is the same as 5 + 5 + 5 , for large numbers this could be a lengthy process, and programming tricks are

required to minimise this. Division is a process of repeated subtractions: 15 / 5 can be performed as the following sequence, 15 - 5 = 10, 10 - 5 = 5, 5 - 5 = 0 , since three subtractions were required,the answer is 3. As with multiplication, programming techniques are needed to reduce the time taken to divide large numbers.

Besides arithmetic operations the ALU or Arithmetic/Logic Unit can perform logical operations between data in memory, and the accumulator. consisting of three instructions AND, OR and EOR. The AND instruction performs a bit by bit logical AND operation between a memory location and the accumulator,storing the result in the accumulator. This operation can be used to reset or mask a single bit or group of bits in a memory location. In the decimal mode each byte holds two digits, the AND instruction can be used to extract one digit. Where there is a zero in the operand, there is a zero in the result. To mask out the most significant decimal digit stored in the bottom four bits, the accumulator is ANDed with 00001111 or hexdecimal 0F.

LDA 25	A9 25	(load the accumulator with decimal 25)
AND 0F (hex)	29 F0	(AND the accumulator with 00001111 binary)
STA 10	8D 0A 00	(store the result in location 10)

On running this program location 10 will contain 05, the 2 being masked out and replaced by a 0.

An OR instruction performs a binary OR on a bit by bit basis between the contents of the accumulator and a memory location, the result is stored in the accumulator. The main use of this instruction is to set a bit or group of bits in a memory location, a logical 1 in the operand field produces a 1 in the corresponding bit of the result. The EOR or Exclusive OR instruction is identical to the OR, except that a logical 1 appears in the result only if there is a 1 in the operand field, and a 0 in the accumulator for the corresponding bit. The main use of the EOR instruction is to produce the two compliment of a byte.

The Processor Status Register and the use of Flags.

The processor status register occupies a very important position in the system architecture of the 6502. It is an eight bit programmable register, unlike the other registers, its function lies between the control and register section of the processor. It is the only register which actually affects the control logic. Seven of the eight bits are used, and each bit, or flag,

has a specific function. Since they are very important it is worthwhile looking at these flags in greater detail.

Flags fall into three catagories, those controllable only by the programmer, those controllable by both programmer and processor, and lastly those controlled solely by the processor. Only one flag falls into the first catagory, the Decimal mode or D flag, occupying bit three of the status register. This flag controls whether the processor performs binary or decimal arithmetic. It can be set by a SED instruction, after which all arithmetic is performed in the decimal mode, until the D flag is cleared by a CLD or clear decimal mode instruction.

Three flags fall into the second category: Carry, Overflow and Interrupt disable. The Carry or C flag is located in bit 0 of the status register, it is modified either by the results of certain arithmetic operations or by the programmer. The carry is also used as a nineth bit during arithmetic operations or by the shift and rotate instructions. The instruction used to set the carry flag is SEC, it can be cleared by CLC. The overflow or V flag occupies bit six of the status register, and is used during signed binary arithmetic to indicate that the result was of greater value than could be contained within the seven bits of the signed byte. The V flag has the same meaning as the carry flag, but also indicates that a sign correction routine must be used if this bit is "on", since the overflow will have erased the sign in bit seven. The programmer can only clear the V flag, using the CLV instruction. The interrupt disable, I flag, controls the operation of the microprocessor interrupt request input and is located in bit two of the status register. Interrupts as seen in Chapter 1 play a very important part in the PET's design, and each time there is an interrupt the I flag is set by the processor. This stops the processor being interrupted by more pulses on the IRQ line, until the interrupt handling program has been completed with a return from interrupt instruction clearing the I flag. The I flag can also be set by the programmer with an SEI instruction if for some reason he wants to prevent the processor being interrupted, as during a precision timed loop subroutine. At the end of such a program the interrupt line can be returned to its normal function by clearing the I flag with a CLI instruction.

The last three flags: Zero, Negative and Break, are controlled solely by the processor. The Zero and Negative flags are either set or reset by nearly every processor operation. The Zero or Z flag is set by the processor whenever the result of an operation is 0, as when two numbers of the same value are subtracted from each other. The Negative or N flag is set equal by the processor to bit seven of the result of an operation.

One of its primary uses is during signed binary arithmetic, if the N flag is set then the result is a negative number. The break or B flag is set by the processor during an interrupt service sequence. The Z flag occupies bit one, the N flag bit seven and the B flag bit four of the status register.

The status register contains seven status bits or flags, each having its own meaning to the programmer at a particular point in the programe. Although the carry and overflow flags are used in arithmetic operations the major use of flags is in combination with the conditional branch instructions. This gives the programmer the capability of incorporating decision making instructions within a program. To test a flag, and, depending on the state of that flag, take one of two courses of action. A conditional branch is functionally the same as the IF... THEN GOTO... statement in Basic, there are a range of these instructions performing different functions and testing different flags. Anyone writing a machine code pragram must keep track of the expected state of all flags at every instant throughout the program. Failure to do this is one of the commonest causes of a program not working or producing the wrong result. An example would be failure to clear the carry flag before an addition, on odd occasions it would have been set by a previous instruction, and thus give rise to erroneous results.

Branches, Jumps and the Program Counter

To understand the use of branch and jump instructions the concept of program sequencing must be understood, and its control by another of the processor registers, the program counter. Figure 2.1. shows the program counter, or PC, as two eight bit registers. Like the other registers they communicate with the data bus, but the outputs are also connected to the sixteen address lines of the processor. One of the PC registers is connected to the bottom eight address lines and is called PCL, the other which is called PCH is connected to the eight high address lines. Although two eight bit registers, they function like a single sixteen bit register. It is the program counter which controls the addresing of memory by being a program or data address pointer, as such it contains the address of the next memory location to be accessed.

At the beginning of a program the PC must contain the address of the first instruction. This is one of the functions of the operating system reset software, it is also performed by the SYS and USR commands when entering a machine code program from Basic. The instruction fetched from memory is stored in the instruction register, to be decoded by the control logic. This process takes one clock cycle, during which time the

program counter is incremented by one to point to the next memory location. The processor usually requires more than one byte to interpret an instruction, this first byte contains the basic operation and is known as the OP CODE. The following one or two bytes, known as the OPERAND, contain either a byte of data or the address of the data on which the operation will occur. An instruction may require up to three sequential memory locations, the program counter first points to the OP CODE which is fetched from memory and stored in the instruction register. The PC is incremented and points to the next memory location, the contents of which are fetched and stored in the ALU, in a three byte instruction this will be the low order address of the data. The program counter is again incremented and the high order address fetched from the third memory location. The processor then latches the two bytes of the address onto the address bus via the ALU, fetches the data, and performs the operation. Having completed the operation, which usually takes about four clock cycles, the processor increments the program counter to point to the next instruction and the process is repeated. In this manner the program counter will continue to advance until it reaches the maximum memory location, fetching instructions and addresses.

A sequential program would lack a feature fundamental to computing, the ability to test the result of an operation, and implement various options based on the results of the test. Firstly flags can be used to test the result of an operation, secondly the contents of the program counter must be changed to point to the start of a new program. The simplest way of changing the contents of the program counter is with the JMP or Jump to new location instruction. This as its name implies does not perform any tests on the results of a previous operation. It simply loads a new sixteen bit address into the program counter thereby forcing the processor to start operating at the new address.

There are eight different conditional branch instructions, they can be divided into four groups, each testing the state of one of the status register flags. The four flags tested by the conditional branch instructions are: Carry, Zero, Negative and Overflow, one instruction tests if the flag is set, and the other if it is clear. The two instructions for the Carry flag are BCC or Branch on Carry Clear and BCS or Branch on Carry Set. The Operand contains the address to which the program jumps if the condition being tested is true. The addressing mode used is unique to conditional branch instructions, it is called relative addressing.

In relative addressing the new address is stored as just one byte, which is added to the current contents of the program counter. To enable the program to branch both forwards and backwards the relative address can be

either a positive or a negative number. The fact that relative branch addresses are stored as a signed single byte limits the maximum size of the branch to either 128 bytes forwards or backwards, this may seem a limitation but in practice it is not.
The eight conditional branch instructions are:

BMI - Branch on Reult Minus
Testing the N flag
BPL - Branch on Result Plus
BCC - Branch on Carry Clear
Testing the C flag
BCS - Branch on Carry Set
BEQ - Branch on Result Zero
Testing the Z flag
BNE - Branch on Result Not Zero
BVS - Branch on Overflow Set
Testing the V flag
BVC - Branch on Overflow Clear

Most operations involve the setting of one or more flags, but a small group of test instructions are specifically designed to set flags for testing by a branch instruction. The most commonly used is the Compare Memory and Accumulator or CMP instruction. It allows the programmer to compare a value in memory to one in the accumulator without altering the value in the accumulator. If the two values are equal the Z flag is set, otherwise it is reset. The N flag is set equal to bit 7 and the carry flag is set when the value in memory is less than or equal to that in the accumulator. The BIT instruction tests single bits in memory with the corresponding bits in the accumulator.

Addressing Modes

At this stage it is a good idea to look at the various addressing modes used by the processor, so far we have met only absolute and relative addressing. There are thirteen different addressing modes and most instructions can be performed in more than one mode. The LDA instruction can use one of eight different modes of addressing. The simplest mode is implied addressing which is used exclusively by single byte instructions operating on the internal processor registers. In an instruction like CLC (Clear Carry) no data is accessed therefore no address is required. It is implied that a register, in this case the Status Register is to be operated upon. Immediate addressing is used whenever the programmer wants to perform an operation using a constant. To put a value of, say 25, in the accumulator we would use the LDA instruction in the Immediate mode. This form of addressing was used in the examples of the operation of arithmetic and logical instructions, data

being stored in the byte immediatly following the OPCODE.

Neither the Immediate or Implied addressing modes use a memory address where data is stored, and are of little use in operations with variables. To address any location in memory would require a full sixteen bit or two byte address stored in the operand part of the instruction. This address points to a memory location where the variable upon which the operation being performed is currently located, or is to be stored. This form of addressing is known as Absolute addressing. A shortened form of absolute addressing can be used when the memory location being accessed lies on page zero of memory. This is the only case where the concept of paging has any importance in the 6502, page zero is just the bottom 256 memory locations. This is called Zero Page Addressing, and uses a single byte address to point to the location of data within page zero. It is a two byte instruction therefore much faster than absolute addressing, it is thus good practice to store all variables in page zero. The remaining non-indexed addressing mode is Relative addressing already met with in conditional branch instructions.

The Index Registers and Indexed Addressing

So far, none of the instructions looked at have accessed more than one byte of data,since the operand field contains a fixed address. This poses problems if ccessing a sequential block of data such as a table or an input buffer. One method would be to use a string of load instructions in the form, load data from address 1 - perform operation - load data from address 2 - perform operation and so on. This is obviously highly wasteful of memory space, it would be more efficient if this program was written as a loop. To do so would require that the address stored as the operand field of the load instruction is incremented each time the program goes round the loop. In this way the operand address will always be pointing to the next byte of data to be accessed. This method is useful, but, execution time is considerably greater than in the straight line programming technique, also it is often undesirable to use a self modifying program.

A more sophisticated approach is the use of a counter, the contents of which are automatically added to the address in the operand field of the instruction. Such a counter is called an Index register. There are two index registers in the 6502, both are eight bit registers, labelled X and Y. They are used by instructions in one of the indexed addressing modes. The simplest is absolute indexed addressing, in this mode the contents of one index register is added to the address in the operand field of the instruction, giving

a new address from which data is to be accessed. The fact that the Index registers are only eight bit registers limits the maximum size of data block accessed using indexed addressing to 256 bytes. In practice the majority of tables are shorter and it is not a significant limitation.

The index registers are controlled and manipulated by a range of special instructions. A number can be loaded to, or stored from the index register and a memory location, by the LDX, LDY and STX, STY instructions. Similarly the contents of the index registers can be compared with a value in memory to test if a conditional branch should take place by using the CPX and CPY instructions. The contents of an index register is changed to point to the next address by incrementing or decrementing it by one. To count up, the instruction used is INX or INY, to count down, DEX or DEY. The remaining index register instructions allow the transfer of the contents of the accumulator into one of the index registers and vice versa. TAX and TAY transfer the accumulator contents into X and Y registers respectively and TXA, TYA transfer the index register contents to the accumulator.

In some programs it may be necessary to have a computed address rather than a base address with an offset, as in absolute indexed addressing. This is done using indirect addressing, instructions in this mode have just a single eight bit address field which points to the effective address as two bytes in page zero. The data address is thus not stored directly in the operand field of the instruction but, indirectly in page zero, all the indirect acesses are indexed except for the JMP instruction. Two modes of indirect addressing are possible, indexed Indirect and Indirect Indexed Addressing.

In Indexed Indirect addressing index register X is added to the operand zero page address. This points to locations where the sixteen bit data address is stored. One of the major uses of this addressing mode is in retrieving data from a table or list of addresses, as in polling I/O devices or performing string operations. In Indirect Indexed addressing the sixteen bit address pointer in page zero is first accessed then offset by the contents of index register Y to give the true data address. The location of the pointer is fixed, whereas in the indexed indirect mode it is variable being offset by the contents of index register X. Indirect indexed addressing combines the advantage of an address that can point anywhere in memory with the offset capability of the index register. It is a particularly powerful method of accessing the nth element of a table, providing the start address is stored in page zero.

The Stack Register and its Use.

The stack register is the last of the processor registers, and is mainly concerned with the handling of interrupts and subroutines. It is an eight bit register, its function is identical to that of the program counter since it is an address generator. It is used to point to an address in page 1 of memory,(locations 256 to 511), known as the stack. The stack is a set of memory locations starting at 511 and filled downwards from that location with a maximum size of 255 bytes. It is organised as a LIFO or last in first out structure, which means that the last byte of data stored on the stack is the first byte to be accessed. Every time data is pushed onto the stack the stack pointer is decremented by one, and each time data is pulled off the stack, the stack pointer is incremented by one. The addressing of the stack is independent of the program and based purely upon chronological events. The stack is used as a temporary data store, the most common data being re-entrant addresses generated by subroutines and interrupts. Every time a subroutine is called in a machine code program the current contents of the program counter is saved. On returning from the subroutine the program can be re-entered at the correct location. Similarly every time the processor is interrupted the current address in the program counter is saved before the processor performs the interrupt servicing routine. A subroutine may call other subroutines, requiring the storage of several re-entrant addresses in the stack. The last re-entrant address stored is the first address reloaded into the program counter at the end of the subroutine, hence the LIFO structure of the stack. The calling of subroutines by other subroutines is termed "subroutine nesting" and is a common occurrence in machine code programs. The size of the stack in the 6502 limits the user to 127 levels of nesting, usually far more than is needed.

A subroutine is called by a JSR or Jump to Subroutine instruction. This pushes the current contents of the program counter onto the stack. A location stored as the operand field is then loaded into the program counter. This causes the processor to jump to a new section of the program and start execution from the location in the program counter.

The return from a subroutine to the main program is accomplished by the RTS or Return from Subroutine instruction. This loads the return address from the stack into the program counter. It also increments the program counter to point to the instruction following the JSR. The stack pointer is also incremented to point to the next subroutine address if any.

The stack can be used by the programmer as a temporary storage location for data passed to a

subroutine. The programmer needs a set of instructions to allow him to put data onto the stack and read it back. The current contents of the accumulator can be transferred to the next location on the stack by the PHA or Push Accumulator onto Stack instruction. Data can be read from the current location pointed to by the stack pointer, into the accumulator, by the PLA or Pull Accumulator from Stack instruction. Both instructions automatically cause the stack pointer to be incremented or decremented by one. An example of data storage in the stack is saving the contents of the processor status register when a subroutine is called. The contents of the status register can be pushed onto the stack by the PHP - Push Processor Status on Stack instruction. Then transferred from the stack back to the status register by the PLP - Pull Processor Status from Stack instruction.

It has been assumed in the first part of the chapter that the stack pointer points to a fixed location, automatically incremented or decremented by the processor. But to use the stack pointer the programmer has to be able to change its contents. The stack pointer is loaded by transferring the contents of the X index register to the stack pointer with a TXS - Transfer Index X to Stack Pointer instruction. This instruction is used at the beginning of a program to initialise the stack pointer, it is performed automatically on the PET as part of the power up reset routine. Re-initialising the stack on the PET could cause problems, frequently resulting in a crash and should thus be avoided. The current contents of the stack pointer can be read by loading it into the X index register with a TSX - Transfer Stack Pointer to Index X instruction.

Interrupts

The processing of interrupts is fundamental to the operation of the PET system. As seen in chapter 1 all I/O is interrupt driven, a knowledge of interrupts is thus required by anyone using the user port or the other I/O. There are three input lines which can cause the processor to halt on completion of the current instruction. Store the program counter on the stack and branch to an interrupt servicing routine at an address pointed to by the contents of one of the interrupt vectors. These three lines are Reset, Interrupt Request and Non-Maskable Interrupt (NMI is only implemented on the new dynamic PET). The reset line is only used when the machine is powered up, therfore not of much interest since it is not under user control. It is the two interrupt request lines which are of major interest, for not only is the IRQ the source of all system interrupts, but both lines can also be controlled by the programmer.

The only way a programmer can change the sequence of operations is to load a new address into the program counter. If this were true then an external event could not effect the program sequence, unless the program was written to periodically check for an input. Most inputs are asynchronous, meaning that for an input to occur at the same time as the program is checking for inputs is extremely unlikely. If an input pulse occurred just after an input check, then not until the next check would that pulse be input to the computer. During the interval between checks data at the input may have changed resulting in the loss of information. To overcome such a data loss the processor could be programmed to wait for the data, but this would mean the processor spending most of its time doing nothing.

Interrupts are used to solve this problem, by having a special line signal the processor whenever an input occurs. This considerably simplifies programming, making it unnecessary to repeatedly use an input testing subroutine or have the computer wait for an input. The two interrupt lines used to signal to the processor that an input is present are the IRQ line and the NMI line. By pulling an interrupt line low for at least 20 microseconds an input device can signal that it wishes to send data to the processor. This forces the processor to finish its current instruction, store the program counter and status register on the stack and jump to a memory location pointed to by the interrupt vector. There are two interrupt vectors that for the IRQ line is located at 65,535 and 65,536, for the NMI line at 65,531 and 65,532. The processor could be interrupted again before it was able to retrieve data from the first input. To prevent this the programmer can disable the IRQ line and prevent further interrupts by setting the I flag in the processor status register. This is done by the first instruction in the interrupt handling subroutine, SEI-Set Interrupt Disable. A CLI - Clear Interrupt Disable instruction clears the I flag and allows the processor to be interrupted as normal. Having obtained data from the input the interrupt software can process it for use by the main program or respond with an output from an I/O port. Control is returned to the main program by the RTI-Return from Interrupt instruction. This pulls the contents of the processor status register and program counter off the stack restoring the processor to its pre-interrupt state.

The PET has six sources of interrupt, two from each of the three peripheral I/0 chips, any one of them can interrupt the processor. Since all interrupt lines are tied together giving a single IRQ input to the processor, a means of finding out which device produced the interrupt is needed. This can be done by hardware, but on the PET is done by software, using an interrupt polling routine. This simply means that the interrupt

software tests each of the I/O devices in turn to find out which device generated the interrupt. The I/O devices are tested in fixed order of priority, the highest priority device being tested first and the lowest last. The purpose being that if two devices generate interrupts at the same time then the processor looks at the highest priority, the most important, device first. The scan interrupt in the PET has highest priority, except when using the cassettes when the read interrupt is highest. Each I/O chip has two interrupt inputs and one output connected to the IRQ line. An interrupt from an external device sets either bit 6 or bit 7 of the peripheral I/O chip status register. It also generates the interrupt to the processor. To test which device generated the interrupt the computer simply reads the contents of each of the I/O status registers testing for bit 7 being set. Having determined which device caused the interrupt the appropriate program can be performed.

An interrupt sequence can also be generated by the programmer without an input being present in the IRQ line, by use of the BRK - Break command. This instruction performs a software interrupt and causes program control to be transferred to the address stored in the interrupt vector. The main use of this instruction is in debugging a program, however since it calls one of the interrupt routines its use on the PET is not recommended. For PET users a similar function is provided in the machine code monitor with none of the attendant problems of the BRK instruction.

Data Modify Instructions

A small group of instructions remain which have not been looked at, they are not associated with any particular processor register and are classified as read/modify/write instructions. They all read data from a memory location or accumulator, modify it in a particular way and store the modified data back into memory or the accumulator. These instructions perform four different data modifications, shift, rotate, increment and decrement. A shift instruction is one which takes the contents of the accumulator or a memory location and shifts all bits one bit to the left or right. An example is the LSR-Logical Right instruction, here the data in the accumulator or memory is moved one bit to the right, bit 0 is placed in the carry flag and bit seven set to zero. Similarly the ASL-Arithmetic Shift Left instruction moves the data one bit to the left, bit seven is stored in the carry flag and bit 0 set to zero. Repeated shifts in the same direction will eventualy result in the entire byte being set to zero. Herein lies the difference between a shift and a rotate instruction. In a rotate instruction the contents of the

carry flag is stored in the bit emptied by the shift, thus no data is lost in a rotate instruction. The ROL-Rotate Left instruction shifts the contents of the accumulator or addressed memory left 1 bit with the carry stored in bit 0 and bit 7 stored in the carry flag. With ROR-Rotate Right instruction the data is shifted right 1 bit with bit 0 shifted into the carry and the carry shifted into bit 7. The shift and rotate instructions have a unique form of addressing, in addition to the normal forms and known as accumulator mode addressing. It indicates that the instruction is to operate on the accumulator rather than on a memory location.

Besides shift and rotate the contents of a memory location can be incremented or decremented. INC-Increment Memory by One adds one to the contents of the addressed memory location. DEC-Decrement Memory by One subtracts one in twos compliment form from the contents of the addressed memory location. The main use of increment and decrement is with counters such as table pointers.

Machine code on the PET

A great advantage of the PET over other small micro computer systems is that it can be programed in both Basic and machine code. This gives the programmer the powerful option of using machine code subroutines in a Basic program. The PET normally runs in the Basic mode and there are five ways of accessing the machine code environment. The first two use commands in Basic, these are, USR and SYS. Both commands access a machine code subroutine whose address is specified in the command or in a specific page zero location. The next three methods involve adding machine code subroutines into the operating system. The first being to add a program into the interrupt servicing routines, these are called sixty times a second by the scan interrupt signal. This method allows for example, the scanning of I/O ports for an input, or selectively disabling certain keys on the keyboard. Any situation where a program must be run concurrently with the main program could use this method. The second methods involves inserting extra code into the CHARGOT subroutine which gets each line of Basic from memory prior to its execution by the interpreter. By intercepting each line of Basic before it is executed new Basic instructions can be added. The instruction being performed by a user written machine code subroutine. Both the method of inserting code into the interrupt routine and the addition of extra code into the CHARGOT subroutine will be dealt with in full later on. Lastly, on new ROM machines the NMI line can be used to force the computer to jump to a NMI interrupt handling routine. One use of this is to provide the

machine with a reset button, by connecting a switch between the NMI line and ground a manual interrupt can be generated. To use the reset, the NMI RAM vector (locations 148 and 149) must contain the start address of the monitor. If a program crashes, pressing the reset switch will cause it to jump into the monitor program.

The main reason for using machine code subroutines is that Basic is too slow for many purposes, especially when using the I/O ports. A machine code routine is more than 100 times faster than the same program written in Basic. Another reason for using machine code is that one may want to change the operating system or use some of the operating system subroutines. Thirdly, a reason used by some commercial software producers is that machine code programs can be protected from illegal copying.

The best place to put small machine code programs is in the second cassette buffer, assuming that is that the second cassette is not being used. This 192 byte memory block extends from location 826 to 1018. If the program is longer than 192 bytes or the second cassette buffer is being used then the progam is best located at the top of memory. This area is used by Basic to store character strings and to avoid these overwriting the machine code program the top of memory pointers must be changed. The top of memory pointers are set during power up diagnostics to the highest usable RAM location. By lowering the value of these pointers a block of memory can be reserved exclusivly for use by a machine code program. The operating system will regard the new top of memory pointers as containing the highest memory location usable by Basic. In the old 8K machines these pointers were stored in locations 134 and 135, and in the new machines in 52 and 53. The pointer is stored as the low order byte in 52 (134) and the high order byte in 53 (135). As an example the following commands will lower the top of memory on a 32K machine by 256 bytes:

```
POKE 52,255:POKE 53,126
```

Of the two Basic commands used to call a machine code subroutine, SYS and USR, by far the most powerful and flexible is SYS. With the SYS command one simply specifies the subroutine starting location, thus if it starts at location 826 it can be called with SYS(826). Variables can be transferred between a Basic program and a machine code program by using PEEK and POKE. These read or write single or multiple byte values into memory locations allocated for the purpose and acessed by both programs. Transferring variables in this manner is easier than using the single floating point variable provided for the USR function. It also allows the transfer of more than one variable which USR does not. The only requirement with a SYS subroutine is that the last instruction in the subroutine is a RTS - return

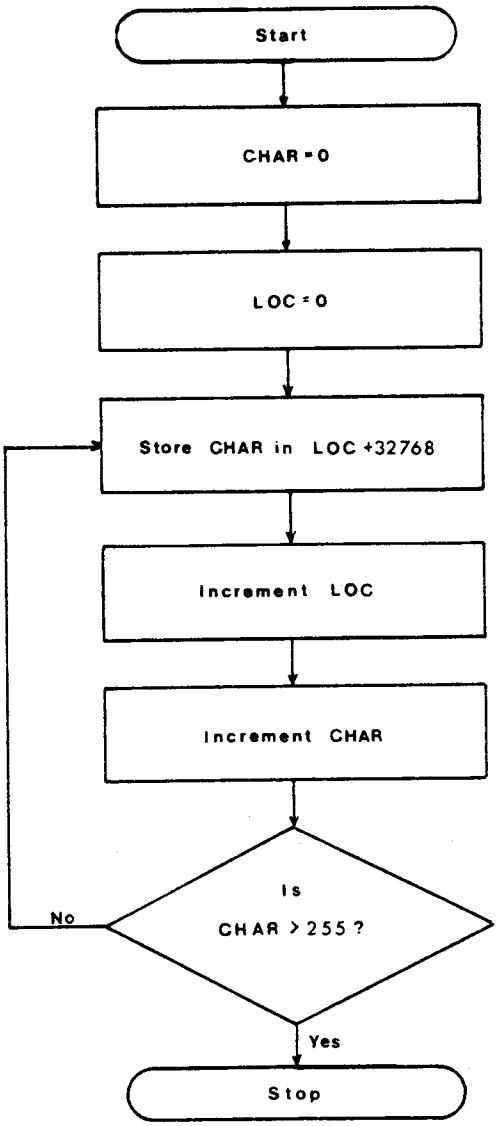
from subroutine since this automatically returns control to the Basic program. Another virtue of SYS is that it is far easier to have more than one machine code subroutine in a Basic program.

The easiest way of entering a machine code program is to incorporate it into the Basic program using a simple loader, to POKE the values byte by byte into the correct locations, you will find several examples elsewhere in this book. Another way is to use the machine code monitor,this is ROM based in the new machines, users of old machines will require a tape version. The monitor allows machine code program to be directly written into memory using hexadecimal code. Also it allows programs to be saved and loaded onto tape in machine code format. Both methods are ideal when writing and entering short - less than 100 bytes - machine code programs, however for longer programs an assembler is essential. An assembler - the Commodore disk based 6502 assembler is highly recommended - allows a program to be written using the mnemonics with lables for variables and jump locations. These are converted by the assembler to binary values which when loaded into memory constitutes the program. Another useful aid to have besides the assembler and monitor is a disassembler. This converts the machine code program back into mnemonics, a function which helps with program fault diagnosis.

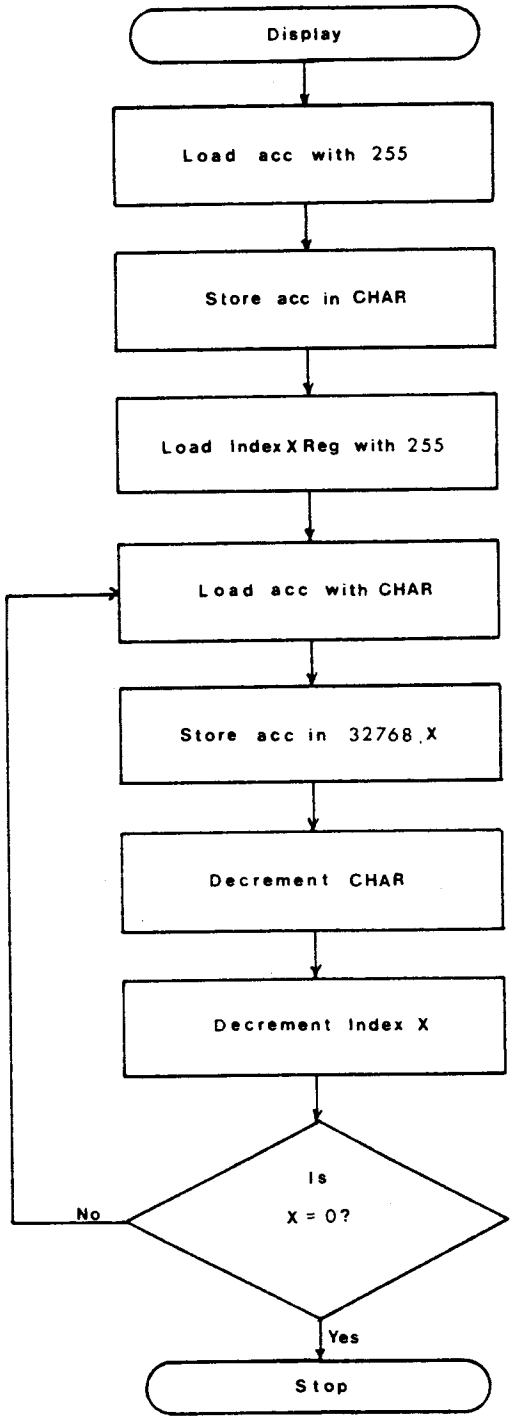
Some techniques for hand assembling and writing machine code programs.

The prospect of writing a machine code program even a small one may seem fairly daunting but providing one uses an orderly and disciplined approach to the problem it need not be hard. A machine code program differs from a Basic program in the approach taken to its writing. Whereas a rough Basic program can be written then polished up by inserting extra lines and changing existing lines. A machine code program must be written as the final version since any changes will require rewriting the whole program. This is because machine code unlike Basic code is dependent on the exact position of instructions in memory. Adding a couple of instructions into the middle of a program will necessitate the changing of all jump, branch and data addresses. This plus a far greater attention to details like current flag status, means that the program must be very carefully planned before it is written. Unless this is done, writing a machine code program will require far greater effort than is necessary and the product far more prone to error.

Stage one in planning a program is to define what the program is required to do, breaking the problem down into a series of steps. To demonstrate this consider the



Initial Version



Final Version

Fig 2.2 Flow Diagrams of Display

following example, to display all the ASCII characters on the screen:

Set LOC to 32768 - set CHAR to zero - store character code CHAR on screen at location LOC - increment CHAR - if CHAR is greater than 255 then all characters have been displayed and program ends, if not then go back and display next CHAR.

From this description we have defined that two variables CHAR and LOC are required, also the program structure requires a loop with a conditional test. For a short program like this a written description is not really required since one can easily remember what one wants the program to do. For longer programs it is an essential part of the process. From the written description one can construct a flow diagram such as the example in Figure 2. The flow diagram can be regarded as a pictorial version of the written description and as a result simpler to follow.

For long programs the flow diagram and written description can get very involved and confusing. It is good practice to split such a program into a series of self contained blocks or subroutine modules. Each module is then treated as a complete program, making program writing and debugging easier. The flow diagram shows the logical pathways through a program and most logical errors can usually be detected at this stage, saving a considerable amount of programming time.

Having drawn a flow diagram the next stage is the construction of a table of variables and locations of system subroutines called. In the example no system subroutines are used but two variables are required:

LOC - pointer to location in screen memory where character is to be stored.
CHAR - Value for ASCII character to be displayed on screen.

It is important that the table contains all variables required, since when writing the program exactly the right amount of space in memory must be left to contain them.

Having defined the logical flow of the program, the variables used and any system subroutines called, a start can be made on writing the program code. Probably the best way is first to draw an expanded versions of the flow diagram. Breaking down each logical step into a series of substeps corresponding to a machine code instruction. In Figure 2 notice that the variable LOC is now stored as the contents of the X index register. Indexed addressing being the easiest way of putting data into successive memory locations. Also the index register (i.e. LOC) is loaded with 255 and decremented,

CODING FORM

PROGRAM DISPLAY

DATE 30/6/79

PAGE 1

ADDRESS MSB	ADDRESS LSB	OPCODE	LABEL	MNEMONIC	AD MODE	OPERAND	FLAGS Z	N	C	I	D	V	CYCLE	COMMENT
03	40	—	CHAR	—										VARIABLE FOR ASCII CHARACTER
	1	A9	DISPLAY	LDA	#	255								START- SET UP LOOP COUNT
	2	FF		/										AND CHARACTER VALUE
	3	8D		STA	ABS	CHAR								INITIALISE CHAR
	4	40		/										
	5	03		/										
	6	A2		LDX	#	255								SET INDEX REGISTER TO 255
	7	FF		/										
	8	AD	NEXTCHAR	LDA	ABS	CHAR								GET CHAR
	9	40		/										
	A	03		/										
	B	9D		STA	ABS,X	$8000,X								STORE AT 32768 + INDEX
	C	00		/										INTO VIDEO MEMORY
	D	80		/										
	E	CE		DEC	ABS	CHAR								PUT NEXT ASCII
	F	40		/										CHARACTER IN CHAR
	50	03		/										
	1	CA		DEX	IMP		X							POINT TO NEXT SCREEN
	2	D0		BNE	REL	NEXTCHAR								LOCATION - LAST CHARACTER?
	3	F4		/										
	4	60	END	RTS	IMP									END & RETURN FROM SUBROUTINE.
	5													
	6													
	7													
	8													

Fig 2.3 Example of Hand Coded Program

rather than 0 and incremented as in the original flow diagram, since it is easier to test for zero than for 255.

For the actual hand assembly and coding of a program it is advisable to use a coding form such as that shown in Appendix B. It helps to considerably reduce the number of errors occuring at this stage. On the first page of the coding form a list of all variables, I/O locations and system subroutine entry points used should be written. Each variable being assigned the number of bytes of memory which it will require. Most will be single byte but some will be two or three byte precision and in the case of character variables or data buffers memory required could be large. When storing a multiple byte numerical variable it is good practice to store the bytes in fixed order, with the least signifiant byte in the first location and the most signifiant byte in the last location. It is easier this way to keep track of which part of a variable is being dealt with. Also index registers can be used to access successive bytes of a variable in the same order that they are processed.

Program variables can be stored in any part of RAM memory not ocupied by either programs or system variables. For maximum speed and reduced program size variables should be stored in page zero of memory, the bottom 255 bytes. On the PET page zero is currently occupied by system variables. This area can be utilised by using two subroutines, one at the beginning of the program and the other at the end. The first disables the system by setting the interrupt flag with an SEI instruction. Then relocates the entire contents of page zero to the top of RAM memory. Leaving page zero free for use by the rest of the program. The last subroutine performs the reverse process, replacing the system variables into page zero prior to re-enabling the system with a CLI instruction. Having decided where variables are to be stored they should be allocated memory locations and the address column on the coding form filled in accordingly.

Using the second expanded flow diagram one can start writing the code onto the coding form using the instruction mnemnics. The first step is to enter the starting location of the program into the address column, then enter the first instruction into the mnemonic column. The addressing mode of the instruction should be entered into the relevant column. This is important since one must be able to calculate how many bytes are required by that instruction, to determine on which line (i.e. at which address) the next instruction should be entered. The label column will contain an entry only if that address is the start of a subroutine or the destination of a jump or branch instruction. On the flow diagram the position of labels is indicated where an operation has more than one entry or exit

point. The label used can be any name but preferably one descriptive of the function of the subroutine or loop. In the example the beginning of the program is given the label DISPLAY and the entry point of the loop is called NEXTCHAR. Entries in the operand column will only be required for instructions referencing other locations in the program and will consist of symbolic labels and variable names. As program code is entered on the coding form the comment column should also be completed. Either with simple references to the flow diagram or a more complete description. At a later date the function and logical flow of the program can thus be easily followed without relying on memory.

Once witten, the program should be checked for logical errors, before being assembled. It will involve less work if errors are detected prior to assembly. The process of hand assembling is done in two stages, the first consists of using the instruction set list to obtain the opcode value for each mnemonic with the specified addressing mode. This hexadecimal value is entered into the opcode column of the coding form on the same line as the mnemonic. If the addressing mode is other than "implied" or "acumulator" then the following one or two bytes will be used to store an address or a value specified in the operand column. If the addressing mode is immediate, then the operand column contains a hexadecimal value which is transferred to the opcode column on the line following that of the instruction code.

The number system used must always be noted, the conventions are that a number prefixed with a % is in binary format, with a $ in hexadecimal format and if no prefix is given then in decimal format. Convention also dictates that an instruction in the immediate mode is identified by a # sign in the address mode column, all other address modes are just an abbreviation of the name. For all other modes the symbol contained in the operand column will correspond to either a lable or variable. If a variable, then the address of the variable can be obtained from the variable table on the first page of the coding form. If the instruction is a jump or branch then the addressing mode used will transfer program control to another section of the program, the operand column will thus contain a label. Since a lable needs the calculation of a jump address it is left until the second part of the assembly procedure. It should be noted that the 6502 requires that all addresses are stored in the form "least significant byte" first, then "most significant byte" thus address 0340 hexadecimal is stored as 40 03.

At the end of the first stage of the assembly process, the opcode column on the coding forms should contain a list of hexadecimal values, one for each location in memory. The exceptions being jump and branch

adresses which are calculated in the second stage. Jump addresses pose no problem since they are stored in either indirect or more commonly absolute mode. Their entries in the opcode column can be obtained from the address of the relavent label. The conditional branch instructions all use relative addressing, where the branch, either forward or backward, is calculated from the location of the branch instruction rather than a fixed location in memory. It is the offset from the current location, which can be up to 127 bytes away, either forward or backward, which must be calculated by the programmer. Great care should be taken with this, any error will cause program control to be transferred to the wrong place, with resultant errors or program crash. To calculate the value for a forward branch one counts the number of bytes from the location of the branch instruction, to the location of the label in the branch operand column, and subtract 2 from this value. If the branch is backwards then the offset is calculated by counting the number of bytes from the branch instruction to the label, then adding 1 and subtracting from 255. The result when converted into hexadecimal can be stored in the opcode column after the branch instruction.

Once all jump addresses have been calculated and a complete list of opcode values obtained the program can be entered into the computer. Before this is done it is advisable to recheck the program, especially the opcode listing for errors (make sure that you can distinguish between 8 and B or A and 4). The opcode listing is best entered into the PET using the machine code monitor - this is the main reason why the opcode was produced using hexadecimal notation. Once entered, the program should be saved before it is run since it is very rarely that a machine code program runs perfectly first time. The contents of memory should then be checked against the opcode listing for any program entry errors, if any are found they should be corrected and the program resaved. One can then try running it. If there is a program error it will probably crash the machine, if so reload the p ogram and the monitor and carefully recheck the logic flow, the coding and the contents of memory. In my experience the three most common causes of fatal program errors are - entry errors, coding errors, and wrongly calculated jump and branch addresses.

The best way of detecting errors is to systematically work through the program inserting a break instruction at points where program failure may have occurred. This will cause the program to return to the monitor, allowing the contents of variable locations to be checked and gradually isolating the fault to a small section of code. Another way of isolating errors is to run the program from different locations, though this does require a careful choice of entry points.

Having detected and removed any fatal errors one may find that the program still does not run properly and produces strange results. Non fatal errors are most commonly caused by either a mistake in the basic logic flow, ignoring the current flag status, using the wrong variable, and quite commonly using the wrong branch instruction.

Successful machine code programming is not hard it requires just a strict adherance to a method and constant attention to detail plus plenty of practice. The methods outlined above should enable PET users to expand their machines capabilities by using machine code subroutines.

THE PET OPERATING SYSTEM 3

Of the 64K addressable memory space on the PET, 14K is occupied by read only memory-ROM. This contains the operating system and Basic software, it extends from address 49152 to 65536 with a gap between 59392 and 61439, locations used by the I/O chips. One can divide this 14K ROM memory area into two parts, one occupied by Basic and the other by the operating system. The area occupied by Basic starts at 49152 and ends at 57623 (in the new ROMs) a total of 8471 bytes. The operating system starts at 57624 and ends at 65536 less the 2048 bytes used by the I/O, a total of 5864 bytes. The purpose of the operating system software is to control system functioning and includes all I/O operations, such as keyboard scanning, display generation, cassette and IEEE input/output, as well as power on reset system initialisation and diagnostics. The Basic routines are solely associated with processing the commands in a Basic program stored in the RAM memory area. They consist of a set of subroutines each capable of executing a specific Basic command.

It is a combination of all the programs stored in this 14K of ROM which allows the user to simply switch on the machine and immediately write or run a program. The structure of the PET's ROM based software is of interest to the user for two reasons. Firstly because it helps to show how the system works. Secondly because many of the subroutines can be used in machine code programs. A knowledge of the location of these subroutines is essential if they are to be used. Unfortunately as most users are aware one is unable to look at any one of the ROM areas using the PEEK command. This is not really a problem since it is easy to examine these areas of memory using the machine code monitor. In this way one can gradually build up a table of the subroutine entry points and deduce the function of the various subroutines. A process aided by relocating sections of the code and dissassembling. The following list has been built up of the major subroutines and their entry points:

System variables memory map(RAM) - old ROM machines

000	$4C constant (6502 JMP instruction)
001-002	USR function address lo, hi

Terminal I/O maintenance

003	Active I/O channel #
004	Nulls to print for CRLF (unused).
005	Column Basic is printing next
006	Terminal width (unused).
007	Limit for scanning source colmns (unused)
008	Line number before storage buffer. (integer address from Basic)
009	$2C constant (special comma for INPUT process).
010-089	BASIC INPUT buffer (80 bytes).
090	General counter for BASIC. (search char ':' or endline)
091	$00 used as delimeter (scan between quotes flag).
092	General counter for BASIC. input buffer pointer.
093	Flag to remember dimensioned variables. 1st char of name .
094	Flag for variable type: 0=numeric; 1=string.
095	Flag for integer type: 80=integer; 00=floating point.
096	Flag to crunch reserved words (protects "& remark).
097	Flag which allows subscripts in syntax.
098	Flags INPUT or READ: 0=Input; 64=Get; 152=Read.
099	Flag sign of TAN.
100	Flag to suppress OUTPUT (+normal;-suppressed).
101	Index to next available descriptor.
102-103	Pointer to last string temporary lo; hi.
104-111	Table of double byte descriptors which point to variables.
112-113	Indirect index #1 lo; hi.
114-115	Indirect index #2 lo; hi.
116-121	Pseudo register for function operands.

Data storage maintenance

122-123	Pointer to start of BASIC text area lo; hi type
124-125	Pointer to start of variables lo; hi byte.
126-127	Pointer to array table lo; hi byte.
128-129	Pointer to end of variables lo; hi byte.

130-131	Pointer to start of strings lo; hi byte.
132-133	Pointer to top of string space lo; hi byte.
134-135	Highest RAM adr lo;hi byte.
136-137	Current line being executed. A zero in 136 means statment executed in a direct command.
138-139	Line # for continue command lo; hi.
140-141	Pointer to next STMNT to execute lo; hi.
142-143	Data line # for errors lo; hi.
144-145	Data statment pointer lo; hi.(145-memory address of data line)

Expression evaluation

146-147	Source of INPUT lo; hi.
148-149	Current variable name.
150-151	Pointer to variable in memory lo; hi.
152-153	Pointer to variable referred to in current FOR-NEXT
154-155	Pointer to current operator in table lo; hi.
156	Special mask for current operator.
157-158	Pointer for function definition lo; hi.
159-160	Pointer to a string descriptor lo; hi.
161	Length of a string of above string.
162	Constant used by garbage collect routine.(3or7 for grbg clct)
163	$4C constant (6502 JMP inst).
164-165	Vector for function dispatch lo; hi.
166-171	Floating accumulator # 3
172-173	Block transfer pointer # 1 lo;hi.
174-175	Block transfer pointer # 2 lo; hi.
176-181	Floating accumulator # 1(FAC#1)(USR function evaluated here).
182	Duplicate copy of sign of mantissa of FAC # 1.
183	Counter for # of bits to shaft to normalize FAC # 1.
184-189	Floating accumulator # 2.(FAC#2)
190	Overflow byte for floating argument.
191	Duplicate copy of sign of mantissa.
192-193	Pointer to ASCII rep of FAC in conversion routine lo; hi.

RAM subroutines

194-199	CHARGOT RAM code. Gets next character from BASIC text.
200	CHARGOT RAM code regets current characters.
201-202	Pointer to source text lo; hi.

203-223	Next random number in storage

OS page zero storage

224-225	Pointer to start of line cursor loc lo; hi.
226	Column positionof cursor.(0-79).
227-228	General purpose start address indirect lo; hi.
229-233	General purpose end address direct lo; hi.
234	Flag for quote mode on/off.
235	timer 1 interrupt sttus: 0=disabled
236	EOT character received
237	character error received
238	current file name length.
239	Current logical file number.
240	Current primary address.
241-242	Current secondary address.(241 device no; 242 max line length)
243-244	Pointer to start of current tape buffer lo; hi.
245	Current screen line #.
246	Data temporary for I/O.
247-248	Pointer to start loc for O.S. lo-hi.(tape start adress/pointer)
249-250	Pointer to current file name lo; hi.
251-254	Tape variable storage.
255	Overflow byte BASIC uses when doing FAC to ACIII conversions.

Page 1

62 bytes on bottom are used for error correction in tape reads. Also, buffer for ASCII when Basic is expanding the FAC into a printable number. The rest of page 1 is used for storage of BASIC GOSUB and FOR NEXT context and hardware stack for the machine.

Page 2

512-514	24-hour clock in 1/60 secs
515	Matrix co-ordinates of last key down (row/col; 255=no key)
516	Shift key status: 0=no shift; 1=shift
517-518	Correction factor for clock, LSB, MSB
519-520	Interrupt driver flag for cassette # 1,switches; # 2 switches (519 for cassette#1 on; 520 for cassette#2 on)
521	Keyswitch PIA duplicate of 59910

522	timing constant buffer
523	Flag # means verify not load into memory.
524	I/O status byte.
525	Index into keystroke buffer.
526	Flag to indicate reverse-field on.
527-536	Interrupt driven key stroke buffer.
537-538	IRQ RAM VECTOR lo; hi.
539-540	BRK instruction RAM VECTOR lo; hi.
541	(IEEE mode)
542	(end of line for input pointer; # characters on screen line)
543	?
544-545	(cursor log row/col, used in input routines)
546	(PBD image for tape I/O)
547	Keyboard input code.
548	Blink cursor flag.
549	Count down to flip cursor. Cursor blink duration.
550	Screen value of input character when cursor moves on.
551	Flag for cursor on/off.
552	(EOT bit received,tape write)
553-577	Table of LSB of start address of video display lines (25).
578-587	Table of logical addresses.
588-597	Table of primary addresses.
598-609	Table of secondary addresses.
608	Input from screen/keyboard flag. 0=keyboard; 1=screen.
610	Index into LA,FA,SA, tables
611	Default input device #.
612	Default output device #.
613	Computation of parity on cassette write.
615-615	?
616	Tape buffer item counter.
617-619	?
620	Serial bit count.
621	Count of redundant tape blocks.
622	?
623	(cycle counter,flip for every bit coming from tape)
624	Count down synchronization or cassette write.
625-626	Index next character in/out tape buffer # 1; # 2.
627	Countdown synchronization on cassette header.
628	Flag to indicate bit/byte error.
629	Flag to indicate tape routine reading shorts.

630-631	Index to addresses to correct on tape read pass 1; pass.
632	Flag for cassette read-tells current function-countdown,read,etc
633	Count of seconds of shorts to write before data.
634-825	Buffer for cassette # 1 (192 bytes)
836-1017	Buffer for cassette # 2 (192 bytes)
1018-1023	Unused.

Subroutine locations in old ROM machines

C000-C091	keyword action addresses
C092-C18F	table of reserved words
C190-C2AB	error messages
C2AC-C2D9	peeks at the stack for active FOR loop
C2AD-C31C	'open up' a space in Basic for insertion of a new line
C31D-C329	tests for stack-too-deep and aborts if found.
C32A-C356	check available memory space
C357-C388	sends a canned error message from C190 area, then drops into:
C389-C391	signals 'ready' (C38B entry for basic warm start).
C394-C3A9	gets a line of input, analyses it, executes it
C3AC-C42E	handles a new line of Basic from keyboard; deletes old line etc.
C430-C460	corrects the chaining between Basic lines after insert/delete
C462-C476	receives a line from the keyboard into the Basic buffer
C479-C48C	gets each character from keyboard
C48D-C521	looks up the keywords in an input line and changes to "tokens"
C522-C550	searches for the location of a Basic line from number in 8,9
C551-C599	implements NEW command - clears everything
C59A-C5A7	sets the Basic pointer to start-of-program
C5A8-C647	performs LIST command
C649-C68F	executes a FOR statement
C692-C6B4	continues to build FOR vectors
C6B5-C6EF	reads and executes the next Basic statement, find next line, etc.
C6F2-C70A	executes the Basic Command as a subroutine
C70D-C71B	performs RESTORE
C71C-C742	handles STOP, END, and BREAK procedures
C745-C75E	performs CONT
C75F-C76D	set pause after carriage return (never called)
C770-C772	performs CLR

C775-C77D performs RUN
C780-C79A performs GOSUB
C79D-C7C9 performs GOTO
C7CA-C7FD performs RETURN
C7FE-C81E scans for start of next Basic line
C820-C840 performs IF
C843-C862 performs ON
C863-C89A gets a fixed point number and stores in 8,9
C89D-C91B performs LET
C91C-C97E check numeric digit/move string pointer
C97F-C982 performs PRINT#
C985-C996 performs CMD
C999-CA24 performs PRINT
CA27-CA41 print string from address in Y,A
CA44-CA76 print a character
CA77-CA9E handles bad input data
CA9F-CAC5 performs GET
CAC6-CADF performs INPUT#
CAE0-CB14 performs INPUT
CB17-CB21 prompts and receives the input
CB24-CC11 performs READ
CC12-CC35 canned messages: EXTRA IGNORED;REDO FROM START
CC36-CC8F performs NEXT
CC92-CCB5 checks Basic format,data type, flags TYPE MISMATCH
CCB8-CD38 inputs and evaluates any expression (numeric or string)
CD3A-CD9C pushes a partialy evaluated argument to the stack
CD9C-CDB9 evaluates a numeric variable, pi, or identifies other symbols
CDBC-CDC0 value of pi in floating binary
CDC1-CDE7 checks for special characters at start of expression
CDE8-CDF6 performs NOT function
CDF7-CE04 performs various functions
CE05-CE0C evaluates expression within parentheses()
CE0B-CE0D checks for right parentheses)
CE0E-CE10 checks for left parentheses (
CE11-CE1B checks for comma
CE1C-CE20 prints SYNTAX ERROR and exits
CE21-CE27 sets up function for future evaluation
CE28-CE39 set up a variable name search
CE3B-CE96 check for special variables,TI,TI$,and ST
CE97-CED5 identifies and sets up function references

CED6-CF05 performs the OR and AND function
CF06-CF6D performs comparisons
CF6E-CF7A sets up DIM execution
CF7B-D00E searches for a Basic variable
D00F-D078 creates a new Basic variable
D079-D087 logs Basic variable location
D088-D098 array pointer subroutine
D099-D09C is 32768 in floating binary
D09D-D0B8 floating point to fixed point conversion for singal values
D0B9-D263 locates and/or creates arrays
D264-D277 performs FRE function
D278-D284 converts fixed point to floating
D285-D28A performs POS function
D28B-D294 checks direct/indirect command, gives 'ILLEGAL DIRECT'
D295-D348 executes DEF statements and evaluates FN(X)
D349-D36A performs STR$ function
D36B-D3D1 scans and sets up string elements
D3D2-D403 builds string vectors
D404-D5C3 does 'garbage collection' -discards unwanted strings
D5C4-D5D7 performs CHR$ function
D5D8-D653 performs LEFT$, RIGHT$, MID$, functions
D654-D662 performs LEN, gets string length
D663-D672 performs ASC function
D673-D684 gets a single byte value from Basic
D685-D6C3 evaluates VAL function
D6C4-D6CF gets two arguments (16 bit and 8 bit) from Basic
D6E6-D701 performs PEEK and POKE
D702-D71D executes WAIT statement
D71E-D890 performs addition and subtraction
D891-D8BE contains floating-point constants
D8BF-D8FC performs LOG function
D8FD-D95D performs multiplication
D95E-D988 loads secondary acumulator from memory ($B8 to $BD)
D989-D9B3 test and adjust primary/secondary accumulators
D9B4-D9E0 routines to multiply or divide by 10
D9E1-DA73 performs division
DA74-DA98 loads primary accumulator from memory ($b0-$B5)
DA99-DACD transfers primary accumulator to memory
DACE-DADD transfers secondary accumulator to primary
DADE-DAEC transfers primary accumulator to secondary

DAED-DAFC	rounds the primary accumulator
DAFD-DB29	extracts primary sign; performs SGN function
DB2A-DB2C	performs ABS
DB2D-DB6C	compares primary accumulator to memory
DB6D-DB9D	Convert Floating point to fixed, unsigned
DB9E-DBC4	perform INT function
DBC5-DC4F	convert ASCII string to floating point
DC50-DC84	get new ASCII digit
DC94-DCAE	print Basic Line number
DCAF-DDE2	convert floating point to ASCII string (at 0100 up)
DDE3-DE23	conversion constants - decimal or clock
DE24-DE2D	evaluation SQR function
DE2E-DE66	evaluation of power function
DE67-DE71	negate (monadic -)
DEA0-DEF2	perform EXP function
DEF3-DF3C	perform function series evaluation
DF45-DF9D	perform RND calculation
DF9E-DFA4	evaluate COS function
DFA5-DFED	evaluate SIN function
DFEE-E019	evaluate TAN function
E048-E077	evaluate ATN function
E0B5-E0CC	Basic scan program, transferred to 00C2-00D9
E0D2-E172	completion of power-on-reset; memory test, etc.
E19B-E1BB	partial test for TI and TI$
E1BC-E1E0	input/read/get director
E1E1-E27C	initialize I/O registers, clear screen, reset subroutines
E27D-E3C3	receive input f om keyboard/screen
E3C4-E3E9	set up new screen line
E3EA-E52F	output character to screen
E530-E5DA	check or and perform screen scrolling
E5DB-E66A	start new screen line
E66B-E67D	interrupt entry
E67E-E683	interrupt return
E685-E73E	hardware interrupt routine: cursor flash, tape monitor, keyboard
E73F-E7AB	convert keyboard matrix to ASCII
E7AC-E7B9	write-on-screen subroutine
E7DE-E7EB	print canned monitor message
F0B6-F1CB	IEEE-488 channel open, test, close
F1CC-F22F	get input character from keyboard, screen cassette, IEEE
F230-F27C	output character to screen, cassette, IEEE

```
F27D-F2A3  restore normal I/O, clear IEEE channels
F2A4-F2AA  abort (not close!) all files
F2AB-F2B7  locate logical file table entry
F2B8-F2C7  transfer file table entries to Device, Command
F2C8-F329  perform file CLOSE
F32A-F33E  test stop key
F33F-F345  test if direct/indirect command for suppressing file advice
F346-F3FE  perform file LOAD
F3FF-F421  print "SEARCHING.."
F422-F432  print "LOADING.." or "VERIFYING"
F433-F461  get parameters for LOAD and SAVE
F462-F494  perform IEEE sequences for LOAD, SAVE, and OPEN
F495-F4BA  search for specific tape header
F4BB-F4D3  perform VERIFY
F4D4-F529  get parameters for OPEN and CLOSE
F52A-F5AD  perform OPEN
F5AE-F5E2  search for any tape header
F5E3-F5EC  clear tape buffer
F5ED-F64C  write tape header
F64D-F666  get start and end addresses from tape header
F667-F67C  set buffer start address
F67D-F694  set tape buffer start and end pointers
F695-F69D  perform SYS command
F69E-F71B  perform SAVE
F71C-F735  find unused secondary address
F736-F78A  update clock
F78B-F7DB  set input device
F7DC-F82C  set output device
F82D-F83A  bump tape buffer counter
F83B-F85D  wait for cassette PLAY switch
F85E-E870  test cassette switch line
F871-F87E  wait for cassette RECORD and PLAY switches
F87F-F8B8  read tape initiation routine
F8B9-F8D1  write tape initiation routine
F8D2-F912  complete tape read or write
F913-F91D  wait for I/O completion
F91E-F92D  test stop key and abort if necessary
F92E-F95E  subroutine to set tape read timing
F95F-FBFB  interrupt routine for tape read
FBDC-FBE4  save memory pointer
```

```
FBE5-FBEB  set ST error flag
FBEC-FBFF  subroutine to count 8 serial bits per byte
FC00-FC1B  subroutine to write a bit to tape
FC1C-FCFA  interrupt 1 for tap write - entry at FC21
FCFB-FD15  terminate I/O and restore normal vectors
FD16-FD37  subroutine to set interrupt vector
FD38-FD47  power-on reset entry; test for diagnostic
FD48-FD7B  diagnostic routine
FD7C-FD8F  checksum routine
FD90-FD9A  pointer advance subroutine
FD9B-FFB1  diagnostic routines
JUMP TABLE:
FFC0       OPEN
FFC3       CLOSE
FFC6       set input device
FFC9       set output device
FFCC       restore normal I/O devices
FFCF       input character (from screen)
FFD2       output character
FFD5       LOAD
FFD8       SAVE
FFDB       VERIFY
FFDE       SYS
FFE1       test stop key
FFE4       get character from keyboard buffer
FFE7       abort all I/O channels
FFEA       update clock
FFED-FFFA  turn off cassette motors
FFFA-FFFB  NMI vector (mangled)
FFFC-FFFD  reset vector
FFFE-FFFF  interrupt vector
```

System variables memory map (RAM) - New ROM machines.

0000-0002	0-2	USR Jump instruction lo-hi
0003	3	General counter for Basic. Search character ':' or endline
0004	4	Scan-between-quotes flag. 00 as delimeter
0005	5	Basic input buffer pointer; # subscripts
0006	6	Default DIM flag. First character of array name
0007	7	Variable flag, type: FF=string, 00=numeric
0008	8	Integer flag, type: 80=integer, 00=floating point
0009	9	DATA scan flag; LIST quote flag; memory flag
000A	10	Subscript flag; FNx flag
000B	11	Flags for input or read, 0==input: 64=get: 152=read
000C	12	ATN sign flag: comparison evaluation flag
000D	13	input flag; suppress output if negative
000E	14	current I/O device for prompt-suppress
0011-0012	17-18	Basic integer address (for SYS, GOTO etc)
0013	19	Temporary string descriptor stack pointer
0014-0015	20-21	Last temporary string vector
0016-001E	22-30	Stack of descriptors for temporary strings
001F-0020	31-32	Pointer for number transfer
0021-0022	33-34	Misc.number pointer
0023-0027	35-39	product staging area for multiplication
0028-0029	40-41	Pointer: Start-of-Basic memory
002A-002B	42-43	Pointer: End-of-Basic, Start-of-Variables
002C-002D	44-45	Pointer:End-of-Variables,Start-of-Arrays
002E-002F	46-47	Pointer: End-of-Arrays
0030-0031	48-49	Pointer: Bottom-of-Strings (moving down)
0032-0033	50-51	Utility string pointer
0034-0035	52-53	Pointer: Limit of Basic Memory
0036-0037	54-55	Current Basic line number
0038-0039	56-57	Previous Basic line number
003A-003B	58-59	Pointer to Basic statement (for CONT)
003C-003D	60-61	Line number, current DATA line
003E-003F	62-63	Pointer to current DATA item
0040-0041	64-65	Input vector
0042-0043	66-67	Current variable name
0044-0045	68-69	Current variable address
0046-0047	70-71	Variable pointer for FOR/Next
0048-0049	72-73	Y save register-new operator save; current operator pointer

004A	74	Special mask for current operator; comparison symbol
004B-004C	75-76	Misc numeric work area; function definition pointer,lo-hi
004D-004E	77-78	Work area; pointer to string description
004F	79	Length of above string
0050	80	constant used by garbage collect routine, 3 or 7
0051-0053	81-83	Jump vector for functions
0054-0058	84-88	Misc numeric storage area
0059-005D	89-93	Misc numeric storage area
005E-0063	94-99	Accumulator#1: E,M,M,M,M,S
0064	100	Series evaluation constant pointer
0065	101	Accumulator hi-order propogation word
0066-006B	102-107	Accumulator#2
006C	108	Sign comparison, primary vs. secondary
006D	109	Low-order rounding byte for Acc#1
006E-006F	110-111	Cassette buffer length/Series Pointer
0070-0087	112-135	Subrtn: Get Basic Char; 77,78=pointer
0088-008C	136-140	RND storage and work area
008D-008F	141-143	Jiffy clock for TI and TI$
0090-0091	144-145	IRQ RAM vector,lo-hi; hardware interrupt vector
0092-0093	146-147	Break interrupt vector
0094-0095	148-149	NMI RAM interrupt vector,lo-hi
0096	150	Status word ST
0097	151	Which key depressed: 255=no key
0098	152	Shift key: 1 if depressed
0099-009A	153-154	Clock correction factor;lsb-msb; 1/30 sec increment
009B	155	Keyswitch PIA duplicate of 59410 : STOP and RVS flags
009C	156	Timing constant buffer
009D	157	Load=0, Verify=1
009E	158	# characters in keyboard buffer
009F	159	Screen reverse flag
00A0	160	IEEE-488 output flag: FF=character waiting
00A1	161	End-of-line-for-input pointer
00A3-00A4	163-164	Cursor log (row,column)
00A5	165	IEEE-488 output character buffer
00A6	166	Key image
00A7	167	0=flashing cursor, else no cursor
00A8	168	Countdown for cursor timing
00A9	169	Character under cursor
00AA	170	Cursor blink flag
00AB	171	EOT bit received

00AC	172	Input from screen/input from keyboard
00AD	173	X save flag
00AE	174	How many open files; pointer into file table
00AF	175	Input device, normally 0
00B0	176	Output CMD device, normally default of 3
00B1	177	Tape character parity
00B2	178	Byte reveived flag
00B4	180	Tape buffer character
00B5	181	Pointer in filename transfer
00B7	183	Serial bit count
00B9	185	Cycle counter
00BA	186	Countdown for tape write; sync on tape header
00BB	187	Tape buffer#1 count
00BC	188	Tape buffer#2 count
00BD	189	Write leader count; Read pass1/pass2
00BE	190	Write new byte; Read error flag
00BF	191	Write start bit; Read bit seq error
00C0	192	Pass 1 error log pointer
00C1	193	Pass 2 error correction pointer
00C2	194	Current function; 0-Scan; 1-15=Count; $40=Load; $80=End
00C3	195	Read checksum; Write leader length
00C4-00C5	196-197	Pointer to screen line
00C6	198	Column position of cursor on above line (0-79)
00C7-00C8	199-200	Utility pointer: tape buffer,scrolling
00C9-00CA	201-202	Tape end address/end of current program
00CB-00CC	203-204	Tape timing constants
00CD	205	Flag for quote mode 0=direct cursor, else programmed cursor
00CE	206	Timer 1 enabled for tape read; 00=disabled
00CF	207	EOT signal received from tape
00D0	208	Read character error
00D1	209	# characters in file name
00D2	210	Current logical file number
00D3	211	Current secondary addrs, or R/W command
00D4	212	Current device number
00D5	213	Line length (40 or 80) for screen
00D6-00D7	214-215	Start of tape buffer, address
00D8	216	Line where cursor lives
00D9	217	Last key input; buffer checksum; bit buffer
00DA-00DB	218-219	Pointer to current file name
00DC	220	Number of keyboard INSERTs outstanding

00DD	221	Write shift word/Receive input character
00DE	222	#blocks remaining to write/read
00DF	223	Serial word buffer
00E0-00F8	224-248	Screen line table: hi order address & line wrap
00F9	249	Interrupt driver flag for cassette#1 status switch
00FA	250	Interrupt driver flag for cassette#2 status switch
00FB-00FC	251-252	Tape start address
0100-010A	256-266	Binary to ASII conversion area
0100-013E	256-318	Tape read error log for correction
0100-01FF	256-511	Processor stack area
0200-0250	512-592	Basic input buffer
0200-0201	512-513	Program counter
0202	514	is processor status
0203	515	is accumulator
0204	516	X index
0205	517	Y index
0206	518	stack pointer
0207-0208	519-520	user modifiable IRQ
0251-025A	593-602	Logical file number table
025B-0264	603-612	Device number table
0265-026E	613-622	Secondary address, or R/W cmd, table
026F-0278	623-632	Keyboard input buffer
027A-0339	634-825	Tape#1 buffer
033A-03F9	826-1017	Tape#2 buffer
03FA-03FB	1018-1019	Vector for Machine Language Monitor
0400-7FFF	1024-32767	Available RAM including expansion
8000-8FFF	32768-36863	Video RAM
9000-BFFF	36864-49151	Available ROM expansion area
C000-E0F8	49152-57592	Microsoft Basic interpreter
E0F9-E7FF	57593-59391	Keyboard, screen, interrupt programs
E810-E813	59408-59411	PIA 1 - Keyboard I/O
E820-E823	59424-59427	PIA 2 - IEEE-488 I/O
E840-E84F	59456-59471	VIA - I/O and timers
F000-FFFF	61440-65535	Reset, tape, diagnostics, monitor

Subroutine locations in new ROM machines

C000-C045	Action addresses for primary keywords
C046-C073	Action addresses for functions
C074-C091	Hierarchy and action addresses for operators
C092-C192	Table of Basic keywords
C193-C2A9	Basic messages, mostly error messages
C2AA-C2D7	Search stack for FOR or GOSUB activity
C2D8-C31A	Open up space in memory
C31B-C327	Test: stack too deep?
C328-C354	Check available memory
C355-C388	Send canned error message, then:
C389-C3AA	Print Ready.
C3AB-C441	Handle new Basic line from keyboard
C442-C46E	Rebuild chaining of Basic lines in memory
C46F-C494	Receive line from keyboard
C495-C52B	Change keywords to Basic tokens
C52C-C55A	Search Basic for a given Basic line number
C55B-C576	Perform NEW, then:
C577-C5A6	Perform CLR
C5A7-C5B4	Reset Basic execution to start-of-program
C5B5-C657	Perform LIST
C658-C6FF	Perform FOR
C700-C72F	Execute Basic statment
C730-C73E	Perform Restore
C73F-C76A	Perform STOP and END
C76B-C784	Perform CONT
C785-C78F	Perform RUN
C790-C7AC	Perform GOSUB
C7AD-C7D9	Perform GOTO
C7DA-C7F2	Perform RETURN, and perhaps:
C7F3-C80D	Perform DATA, i.e., skip rest of statment
C80E-C810	Scan for next Basic statment
C811-C82F	Scan for next Basic line
C830-C842	Perform IF, and perhaps:
C843-C852	Perform REM, i.e., skip rest of line
C853-C872	Perform ON
C873-C8AC	Get fixed-point number from Basic
C8AD-C927	Perform LET

C928-C936 Add ASCII digit to accumulator #1
C937-C98A Continue to perform LET
C98B-C990 Perform PRINT#
C991-C9A4 Perform CMD
C9A5-CA1B Perform Print
CA1C-CA38 Print string from memory
CA39-CA4E Print single format character (space, cursor-right,?)
CA4F-CA7C Handle bad input data
CA7D-CAA6 Perform GET
CAA7-CAC0 Perform INPUT#
CAC1-CAF9 Perform INPUT
CAFA-CB06 Prompt and receive input
CB07-CBFB Perform READ; common routines used by INPUT and GET
CBFC-CC1F Messages: EXTRA IGNORED, REDO FROM START
CC20-CC78 Perform NEXT
CC79-CC9E Check data type, print TYPE MISMATCH
CC9F-CDEB Input & evaluate any expression (numeric or string)
CDEC-CDF1 Evaluate expression within parentheses ()
CDF2-CDF4 Check right parenthesis)
CDF5-CDF7 Check left parenthesis (
CDF8-CE02 Check for comma
CE03-CE07 Print SYNTAX ERROR and exit
CE08-CE0E Set up function for future evaluation
CE0F-CE88 Search for variable name
CE89-CEC7 Identify and set up function references
CEC8-CECA Perform OR
CECB-CEF7 Perform AND
CEF8-CF5F Perform comparisons,string or numeric
CF60-CF6C Perform DIM
CF6D-CFF6 Search for variable location in memory
CFF7-D000 Check if ASCII character is alphabetic
D001-D077 Create new Basic variable
D078-D088 Array pointer subroutine
D089-D08C 32768 in floating binary
D08D-D0AB Evaluate expression for positive integer
D0AC-D227 Find or create array
D228-D258 Compute array subscript size
D259 Perform FRE
D26D-D279 Convert fixed point to floating point
D27A-D27F Perform POS

D280-D28C Check if direct command, print ILLEGAL DIRECT
D28D-D2BA Perform DEF
D2BB-D2CD Check FNx syntax
D2CE-D33E evaluate FNx
D33F-D34E Perform STR$
D34F-D360 Calculate string vector
D361-D3CD Scan and set up string
D3CE-D3FF Subroutine to build string vector
D400-D496 Garbage collection subroutine
D497-D4DF Check for most eligible string collection
D4E0-D516 Collect a string
D517-D553 Perform string concatenation
D554-D57C Build string into memory
D57D-D5B4 Discard unwanted string
D5B5-D5C5 Clean the descriptor stack
D506-D5D9 Perform CHR$
D5DA-D605 Perform LEFT$
D606-D610 Perform RIGHT$
D611-D63A Perform MID$
D63B-D655 Pull string function parameters from stack
D656-D65B Perform LEN
D65C-D664 Move from string-mode to numeric-mode
D665-D674 Perform ASC
D675-D686 Input byte parameter
D687-D605 Perform VAL
D6C6-D6D1 Get two parameters for POKE or WAIT
D6D2-D6E7 Convert floating point to fixed point
D6E8-D706 Perform PEEK
D707-D70F Perform POKE
D710-D72B Perform WAIT
D72C-D732 Add 0.5 to accumulator#1
D733-D744 Perform subtraction
D745-D76D Microsoft joke
D76E-D852 Perform addition
D853-D889 Complement accumulator#1
D88A-D88E Print OVERFLOW and exit
D88F-D8C7 Multiply-a-byte subroutine
D8C8-D8F5 Function constants: 1, SOR(.5),SOR(2), -00.5. etc.
D8F6-D936 Perform LOG
D937-D964 Perform multiplication

D965-D997 Multiply-a-bit subroutine
D998-D9C2 Load accumulator #2 from memory
D9C3-D9DF Test and adjust accumulators #1 and #2
D9E0-D9ED Handle overflow and underflow
D9EE-DA04 Multiply by 10
DA05-DA09 10 in floating binary
DA0A-DA12 Divide by 10
DA13-DA1D Perform divide-into
DA1E-DAAD Perform divide-by
DAAE-DAD2 Load accumulator #1 from memory
DAD3-DB07 Store accumulator #1 into memory
DB08-DB17 Copy accumulator #2 into accumulator #1
DB18-DB26 Copy accumulator #1 into accumulator #2
DB27-DB36 Round off accumulator #1
DB37-DB44 Compute SGN value of accumulator #1
DB45-DB63 Perform SGN
DB64-DB66 Perform ABS
DB67-DBA6 Compare accumulator #1 to memory
DBA7-DBD7 Convert floating-point to-fixed-point
DBD8-DBFE Perform INT
DBFF-DC89 Convert string to floating-point
DC8A-DCBE Get new ASCII digit
DCBF-DCCD String conversion constants: 99999999,999999999, 1E+9
DCCE-DCD8 Print IN, followed by:
DCD9-DCE8 Print Basic line number
DCE9-DE1C Convert number or TI$ to ASCII
DE1D-DE5D Constants for numeric conversion
DE5E-DE67 Perform SQR
DE68-DEA0 Perform power function
DEA1-DEAB Perform negation
DEAC-DED9 Constants for string evaluation
DEDA-DF2C Perform EXP
DF2D-DF76 Function series evaluation subroutines
DF77-DF7E Manipulation constants for RND
DF7F-DFD7 Perform RND
DFD8-DFDE Perform COS
DFDF-E027 Perform SIN
E028-E053 Perform TAN
E054-E08B Constants for trig evaluation: pi/2, 2#pi, .25, etc.
E08C-E0BB Perform ATN

E0BC-E0F8 Constants for ATN series evaluation
E0F9-E110 Subroutine to be moved to zero page ($70 to$87)
E111-E115 Initial RND seed
E116-E1B6 Initialize Basic system
E1B7-E1DD Messages: BYTES FREE, ### COMMODORE BASIC ###
E1DE-E228 Initialize I/O register, and:
E229-E256 Clear screen, and:
E257-E284 Home cursor
E285-E2F3 Input from screen or keyboard; wait for input completion
E2F4-E33E Input from screen
E33F-E34B Test for quotation mark and reverse quote-flag
E34C-E38A Set up screen print parameters
E38B-E395 Prevent 80-character line from getting any longer
E396-E3B3 Extend 40-character line to 80 characters
E3B4-E3D7 Back into the previous line (via DEL or CURSOR LEFT key)
E3D8-E518 Handle ASII character for screen output
E519-E53E Go to next line on screen
E53F-E5B9 Scroll the screen
E5BA-E61A Open a line on the screen (via INSERT key)
E61B-E62D Main interrupt entry point
E62E-E6E9 Hardware interrupt: service clock,keyboard,cassettes
E6EA-E6F7 Print character on screen
E6F8-E769 Table: decoder for keyboard matrix
E76A-E796 MLM subroutine: output hex digits
E797-E7A6 MLM subroutine: swap TMP0 and TMP2
E7A7-E7F6 MLM sibroutine: input hex digits
E7F7-E7FF MLM subroutine: print ?
F000-F0B5 Monitor messages, mostly for Input/Output
F0B6-F0ED Set up IEEE for Talk, Listen etc
F0EE-F127 Send character to IEEE-488 bus
F128-F135 Output character immediate mode to IEEE-488 bus
F136-F155 Send errors: WRITE TIMEOUT, DEVICE NOT PRESENT, etc
F156-F163 Send canned I/O message
F164-F16E Send immediate mode Listen command, then secondary address
F165-F17E Output character deferred mode to IEEE-488
F17F-F18B Drop IEEE channel: send Unlisten or Untalk
F18C-F1D0 Input character from IEEE-488 bus
F1D1-F1E0 GET a character
F1E1-F231 INPUT from any device
F232-F26D OUTPUT a character to any device

F26E-F283 Abort all files, and;
F284-F28C Restore normal I/O devices
F28D-F2A8 Find file table entry; set parameters from file table
F2A9-F300 Perform CLOSE
F301-F30E Test STOP key
F30F-F314 Action STOP key
F315-F31C Send message if direct mode
F31D-F321 Test if direct mode
F322-F3C1 Perform program loading
F3C2-F409 Perform LOAD
F40A-F43D Subroutines: Print SEARCHING...; Print LOADING or VERIFYING
F43E-F45F Get Load or Save parameters
F460-F465 Get a byte parameter
F466-F493 Send program name to IEEE-488 bus
F494-F4B6 Find a specific tape header
F4B7-F4CD Perform VERIFY
F4CE-F50D Get parameters for OPEN, CLOSE
F50E-F515 Abort calling subroutines if end-of-line (default parameters)
F516-F520 Confirm comma, else send SYNTAX ERROR
F521-F5A5 Perform OPEN
F5A6-F5D9 Find any tape header
F5DA-F63B Write tape header
F63C-F655 Get start and end program addresses from tape header
F656-F66B Set cassette buffer address according to device number
F66C-F683 Set tape start and end addresses from buffer address
F684-F68C Perform CMD
F68D-F69D Set tape start and end addresses from Basic pointers
F69E-F728 Perform SAVE
F729-F76C Update TI and TI$, and copy STOP key to work area
F76D-F76F TI constant: limit of clock (24 hours)
F770-F7BB Set input device
F7BC-F805 Set output device
F806-F811 Advance tape buffer pointer (for INPUT#, GET#, and PRINT#)
F812-F834 Wait: PRESS PLAY ON TAPE#
F835-F846 Test if cassette button(s) pressed
F847-F854 Wait: PRESS PLAY & RECORD ON TAPE#
F855-F885 Initiate tape read
F886-F8E5 Initiate tape write
F8E6-F8EF Test for I/O interrupt completion
F8F0-F8FF Test stop key

F900-F930 Set expected timing for next input bit from tape
F931-FA56 Interrupt entry: Read tape bits
FA57-FB75 Store received tape characters
FB76-FB7E Set tape read/write address back to starting point
FB7F-FB83 Flag I/O error into ST
FB84-FB92 Reset 8-counter and flags for a new byte
FB93-FBAE Write a transition to cassette tape
FBAF-FC40 Write interrupt 2: write data to tape
FC41-FC7A Write interrupt 1: Write tape shorts (leader)
FC7B-FC95 Terminate tape: restore normal interrupt vector
FC96-FCA5 Set interrupt vector from table
FCA6-FCB3 Turn off cassette motors
FCB4-FCC5 Perform running checksum calculation
FCC6-FCD0 Check: read/write pointer at limit?
FCD1-FCFD Power on reset entry point
FCFE-FD00 NMI interrupt entry point
FD01-FD10 Table of interrupt vectors
FD11-FFB0 Machine Language Monitor (MLM) - see Commodore documentation
FFB1-FFBF Commodore copyright statement

*****JUMP TABLE*****
FFC0 OPEN
FFC3 CLOSE
FFC6 Set input device
FFC9 Set output device
FFCC Restore default I/O devices
FFCF Input character
FFD2 Output character
FFD5 LOAD
FFD8 SAVE
FFDB VERIFY
FFDE SYS
FFE1 Test STOP key
FFE4 Get character
FFE7 Abort all I/O activity
FFEA Clock update
FFF0-FFF9 Unused
FFFA-FFFF Hardware vectors: NMI, Reset, Interrupt

When the PET is switched on a reset is generated by the system hardware causing the processor to jump to a subroutine whose location is pointed to be the contents of the reset vector. The subroutine called is part of the operating system and performs the functions of testing memory, determining how much space is available and initialising variables in the bottom 634 bytes of memory.

Memory is tested by the simple method of writing a value into a memory location and reading it back again. If the value read back is different the operating system decides it has found the top memory. This feature is useful since it automatically isolates any memory fault giving dropped or transposed bits. It is not able to detect many of the more obscure memory faults or faults in the bottom IK of memory. The highest usable RAM address is then stored in locations 52 and 53 (in old ROM machines 134 and 135). By changing the contents of these locations the user can lower the top of memory to leave space for machine code programs or data stored using POKE. It is the highest RAM address, less the amount of memory used for variables and cassette buffers, a total of 1024 bytes, which is displayed on the screen on system power up. The pointer to the start of user memory is stored in locations 40 and 41 (in old ROM machines 122 and 123). The setting of pointers to the top and bottom of memory is part of system initialisation and is required prior to the system running a Basic program.

All variables required by Basic and the operating system are stored in the lowest 643 bytes of memory. The most commonly used variables, buffers, counters etc, are stored in page zero of memory, the bottom 256 bytes. The reason page zero is used for common variables rather than any other area of memory is that the microprocessors zero page addressing capability is much faster and more efficient in memory usage than addressing to other parts of memory. The location and function of most variables has been determined and is shown in the previous table. A knowledge of these locations is very useful to the PET user since by changing their contents one can change the system's operation. The majority of POKE commands contained in this book are located in this area of memory.

The section of memory not used by system variables is available to the user, on a 32K PET this is from location 1024 to 32768 a total of 31744 bytes. This memory space is however not completely available for program storage being also required for the storage of string and numeric variables. It is no use writing a program 7K long and trying to run it on an 8K PET, this will just result in the operating system giving an out of memory error. The Basic program is stored from location 1025 upwards and the strings and variables are

stored from top of memory downwards. When a program line is entered on the keyboard it is first written into the keyboard buffer. The operating system then transfers it byte by byte as it is entered onto the screen. The line however is not entered into memory until a carriage return is pressed. This causes the operating system to transfer the program line just entered from the screen into memory, where it can be executed with a run command. Each line is stored in a specific format using a compressed version of the Basic text. This reduces the memory requirements of a program and allows longer programs to be run. The compression of Basic text involves conversion of the Basic commands into single byte tokens. The command PRINT instead of being stored as five ASCII characters is stored in a single byte as the decimal value 153. When a program is listed the text compression process is reversed, as far as the user is concerned the program is stored in the same form as it was written.

A useful result of text compression is a shorthand way of writing Basic commands, either in a program or direct command mode. This relies on the fact that the routine which converts commands to tokens looks only at the first two or three characters of a command word. Other characters in the command word are there for the users convenience only. Normally if we entered only the first couple of characters of a command the computer would respond with a syntax error message. This can be done though by using a simple method of fooling the error detection routines. The method used is this, to enter any Basic reserved word type the first letter of the word then depress the shift key and type the second letter. By using just the first two letters there could be confusion between commands which share the first two letters. For example STOP and STEP, in these cases the first two letters should be typed followed by the third with the shift key depressed. The following is a list of Basic commands and their abbreviated form with the numerical value of the command token in both decimal and hexadecimal.

Command	Abbreviation	Decimal token	Hexadecimal token
END · CONT	En	128	80
FOR	Fo	129	81
NEXT	Ne	130	82
DATA	Da	131	83
INPUT#	In	132	84
INPUT ·	INp	133	85
DIM	Di	134	86
READ	Re	135	87

LET	Le	136	88
GOTO	Go	137	89
RUN	Ru	138	8A
IF	If	139	8B
RESTORE	REs	140	8C
GOSUB	GOs	141	8D
RETURN	REt	142	8E
REM	REM	143	8F
STOP	St	144	90
ON	On	145	91
WAIT	Wa	146	92
LOAD	Lo	147	93
SAVE	Sa	148	94
VERIFY	Ve	149	95
DEF	De	150	96
POKE	Po	151	97
PRINT#	Pr	152	98
PRINT	?	153	99
CONT	Co	154	9A
LIST	Li	155	9B
CLR	Cl	156	9C
CMD	Cm	157	9D
SYS	Sy	158	9E
OPEN	Op	159	9F
CLOSE	CLo	160	A0
GET	Ge	161	A1
NEW	NEw	162	A2
TAB	Ta	163	A3
TO	To	164	A4
FN	Fn	165	A5
SPC	Sp	166	A6
THEN	Th	167	A7
NOT	No	168	A8
STEP	STe	169	A9
+		170	AA
-		171	AB
*		172	AC
/		173	AD
AND	An	175	AF
OR	Or	176	B0
=		178	B2
SGN	Sg	180	B4
INT	INt	181	B5
ABS	Ab	182	B6
USR	Us	183	B7
FRE	Fr	184	B8
POS	POs	185	B9
SQR	Sq	186	BA
RND	Rn	187	BB
LOG	LOg	188	BC
EXP	Ex	189	BD
COS	COs	190	BE
SIN	Si	191	BF
TAN	TAn	192	C0

ATN	At	193	C1
PEEK	Pe	194	C2
LEN	Le	195	C3
STR$	STr	196	C4
VAL	Va	197	C5
ASC	As	198	C6
CHR$	Ch	199	C7
LEFT$	LEf	200	C8
RIGHT$	Ri	201	C9
MID$	Mi	202	CA

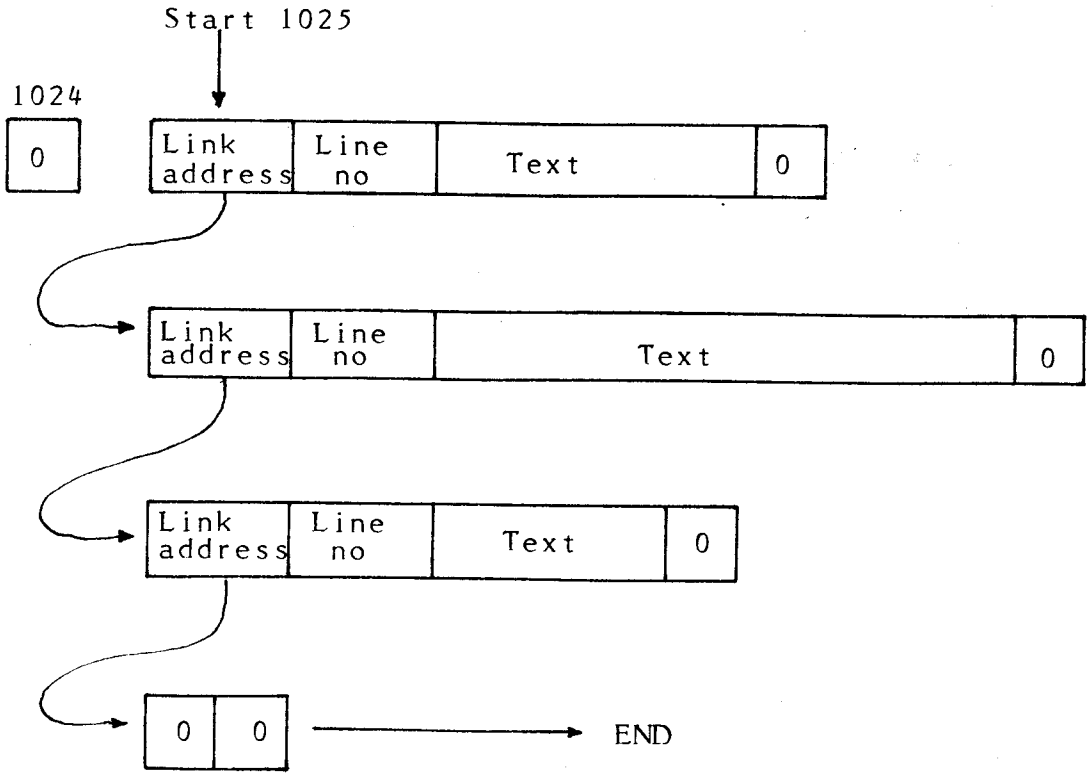
The token value given to a Basic command is a pointer into a table of reserved command words located between 49298 and 49551. By subtracting 127 from the token value the number of the word in that table can be obtained. It should be noted that the technique of using token to represent words can give the programmer a very powerful method of generating print statements without consuming a large amount of memory. This can prove especially useful in games programs, such as Adventure, which require a lot of text generation. By constructing a table of, say, 200 common words each time one of these words appears in a print statement it is represented by a number which points to its location in the table. Obviously some sort of output subroutine is required to convert the token back into a word but the saving in memory space can be considerable.

Having converted the Basic command into a single byte token thereby compressing the Basic text, the line is stored together with the line number and a link address at a location just above that of the last line entered, or if it is the first line at location 1025 upwards. Assumeing that it is the first line of the program which is being entered, then it will be entered into the following locations in the following format:

```
1024 - contents 0
1025 - link address low  }
                         } points to starting
                         } location of next line
1026 - link address high }
1027 - line number low
1028 - line number high
1029 - start of compressed Basic text.
       Number of bytes occupied variable.
       End of line flagged by a zero byte.
```

A Basic program is stored as a series of blocks each of variable length and representing one line in the program. Each block having a fixed format and all blocks being connected via a linked list structure. Each line in a program is stored in memory in the correct position dictated by the magnitude of its line number, thus it will be the line with the lowest number which is stored at the bottom of memory - location 1025 up. The line number is stored in byte 3 and 4 of a block in binary format, this means that the largest line number that can be used in a program is 65535, any number above that will give a syntax error. When a program is run the current line number being executed is stored in locations 54 and 55 (in old ROMs 136 and 137). A direct mode of operation for the processor is indicated when the contents of these two bytes is zero. The double byte link address points to the starting byte of the next line. As each line is executed this address is stored in locations 119 and 120 (in old ROMs 201 and 202), where it is accessed when the operating system fetches the next line. The link address of the last line of a program points not to another link address as in a normal program line, but to two bytes the contents of which are zero.

The storage of a program within memory is best illustrated by the following diagram:



A knowledge of how a program is stored in memory is useful enabling us to perform several operations which the system does not otherwise allow, for example: line renumbering and overlays. Line numbers can be changed simply by changing the contents of bytes three and four of each block (line). The beginning of each line is located using the link address obtained from the previous line. The following is a simple renumbering program, it requires the top and bottom line numbers to be renumbered, the new starting line number and line number increment.

```
60000 INPUT"START AT LINE ";S
60005 INPUT"END AT LINE ";E
60010 INPUT"NEW LINE START";L
60015 INPUT"LINE INCREMENT";I
60020 A=1025:B=256
60030 Q=PEEK(A+2)+B*PEEK(A+3)
60033 IFQ SGOTO60030
60037 IFQ EGOTO60000
60040 POKEA+2,L-INT(L/B)*B
60045 POKEA+3,INT(L/B)
60050 L=L+I:A=PEEK(A)+B*PEEK(A+1)
60055 IFA=0GOTO60000:GOTO60030
```

To use this program first load into the PET then list on the screen, (this is a simple way of merging the renumber program onto the end of the program to be renumbered). The program to be renumbered is then loaded and the renumber program merged with it by placing the cursor over each line on the screen and pressing return. Having done this the renumber program can be run with the command RUN 60000. It should be noted however that this renumber program is very simple and will not renumber any of the jump addresses stored in the Basic text. To do this the program must examine the tokens used in the Basic text area, looking for GOTO or GOSUB commands and renumber their jump addresses. Anyone intending to add this function to the above program should note that whereas the line number is stored in a binary format the jump line is stored in ASCII and is thus of variable length.

Another function which can be performed by manipulating the way a program is stored is creating program overlays. This means calling a program segment from tape or disk and running this program whilst retaining the common subroutines and data used by the previous program segment. By using overlays the programmer can create programs which are much larger than the maximum core size of the machine, without having to manually dump out and reload the data. On the PET, a program can be loaded using the LOAD command within a previous program. If the new program is

shorter, then part of the previous program not replaced by the new program is still retained in memory. But the remaining part of the old program is not accessible normaly by the new program. One can, providing the new program is shorter than the old, use the data generated by the old program in the new program, none of the data areas being affected by loading a new program.

To create an overlay; a) ensure that common subroutines are stored at the end of the old program; b) ensure that the new overlay program is shorter than the old program and does not erase any of the subroutines or data. Lastly a link address must be created between the end of the new program and the start of the subroutines, to replace the end of program marker put there by the operating system. The reason for common subroutines being stored at the end of the old program is that a new program is always loaded into memory starting at address 1024. Thus it is always the lowest line numbers which are replaced. Also the subroutines should have line numbers much higher than any line numbers used in the overlay program. This is because the operating system requires that lines are stored in strict sequence of line number. When new lines are entered, the operating system moves all lines with higher line numbers up in memory, recalculating the link addresses and inserting the new line in the correct position.

Assuming the above criteria have been met, then to link two programs together the location of the two link address bytes of the last line in the overlay program must be known. Also the starting address of the subroutines in the original program must be determined. The following program can be used to find these locations and their contents, it can be merged onto the end of a program using the same method used in the renumber program.

```
60000 INPUT "LINE TO BE EXAMINED";L
60010 A=1025:B=256
60020 Q=PEEK(A+2)+B*PEEK(A+3)
60030 IFQ=LTHENGOTO60060
60040 A=PEEK(A)+B*PEEK(A+1)
60050 GOTO60020
60060 PRINTA;PEEK(A),A+1;PEEK(A+1)
60070 PRINT:GOTO60000
```

By running this program we can look at a particular line number and determine the location and contents of its link address. The program gives the line number, the starting location in memory of that line, plus the contents of the two link address bytes. Using this routine with the original program the start address of the subroutines can be found. The link byte location of the last line of the overlay can also be found using

this program. To connect the two program segments, the start address of the subroutine segment is loaded into the link bytes of the overlay, using POKE commands as follows, X is the start address and A the link byte location:

POKE A,X - INT (X/256)*256: POKE A+1, INT (X/256)

The two programs will now run as one, providing no lines are entered or deleted, this will require the addresses to be recalculated. The technique of altering the link addresses can be used to produce some other interesting ideas, such as making sections of a program unlisted and unrunnable to anyone who does not have the key, where the key consists of a link address which must be inserted into the correct location. Thus for example a commercial software vendor can add an undetectable line of code to a program containing a unique number used to indentify that program and prevent illegal copying.

Data storage.

The entire area of memory not used for program storage is available for storage of data. Firstly, it is worth looking at the simplest form of data storage - using data statements. A data statement is stored as part of a program in the Basic text area of memory. The data is accessed by the program using the READ command. Data stored in data statements though can only be added to by adding program lines. Another limitation is that data can only be accessed from data statements in a serial mode. This means that to find one particular item the whole table of data must be read. The pointer to the current data statment is stored in locations 62 and 63 (in old ROMs 144 and 145). Manipulation of the contents of these locations could provide the user with a means of overcoming the serial search limitation.

Data not stored within the program as data statements, is stored by the program in the area of memory above the Basic text area, as variables. Variables can be divided into two groups, simple variables of the kind used in the following statement; LET X=47 where X is a simple variable. Secondly array variables which are defined by a DIM statement and contain more than one value. The number of values being determined by the number of elements in the DIM statement. For both groups of variables there are three types of data, these are: - real or floating point numbers - integer numbers - and character or string variables, where words are being stored rather than numbers.

Simple variables of whatever data type are stored immediately above the Basic program text area at an address pointed to by the contents of locations 42 and

43 (in old ROMs 124 and 125). The amount of memory used to store these variables depends on the number of variables used by a program. Each variable occupies seven bytes of memory and the next free location in the simple variable storage area is pointed to by the contents of locations 44 and 45 (in old ROMs 126 and 127).

The array variables are stored above the simple variables and thus start from the location pointed to by 44 and 45. The amount of memory used to store the array variables depends on the number of array variables the number of elements in each and the data type of each variable. The top of the storage area used for array variables, which is also the beginning of the unused storage area of memory, is pointed to by locations 46 and 47 (in old ROMs 128 and 129). Since array variables are stored directly above simple variables, whenever a new simple variable is encountered in a program the operating system shifts the entire array variable storage area up seven bytes in memory, thereby opening up a space to accomodate the new variable. This dynamic re-allocation of data storage space is one of the reasons why a machine code subroutine can not be stored in unusued memory space, unless placed above the address stored in the top of memory pointers in locations 52 and 53 (in old ROMs 134 and 135). The re-allocation of memory space slows down a program since every time a new variable is encountered processing stops while the data is moved. When processing speed is important, such as in real time applications, this rather inconsistent variation in speed can be a problem. It is overcome by initialising all the variables - using dummy constants if necessary - at the beginning of the program.

Single value variables are divided into three distinct data types, each being stored in a different format. The only thing all three have in common is that each variable stored requires seven bytes of memory. Both integer and floating point numbers stored as single value variables have both the name and the value stored within the seven bytes allocated to each variable. An integer variable is distinguished from a floating point variable by adding 128 to the ASCII value of the variable name. The formats used are:

INTEGER VARIABLES

first character in variable name (the ASCII value + 128)	second	high order byte of binary representation of integer value	low	0	0	0

FLOATING POINT VARIABLE

first second character in variable name	binary exponent + 129	binary mantissa in packed BCD giving eight digit precision. First bit of first byte is sign bit.

From this, one can see that there is no saving in memory usage by using single value integer variables instead of floating point variables. When the data being stored consists of a string of alphanumeric characters then the variable is stored using the character format. In this format the data is not stored within the seven bytes allocated for variable storage. What is stored is a pointer to an address in memory where this string of characters is stored. Character strings are in fact stored in an area right at the top of memory and extending downwards towards the area occupied by the array variables. By using this method string variables need not be of a fixed length thereby considerably reducing the amount of memory needed to store them. The format used for a string variable is:

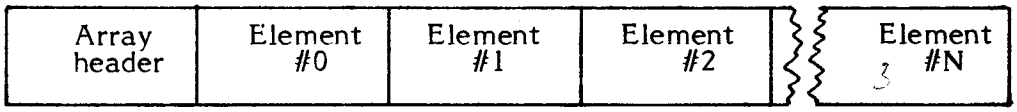
STRING VARIABLES

first second character in variable name, 128 added to ASCII value of second character only.	number of characters	low high order byte of address where string is stored	0	0

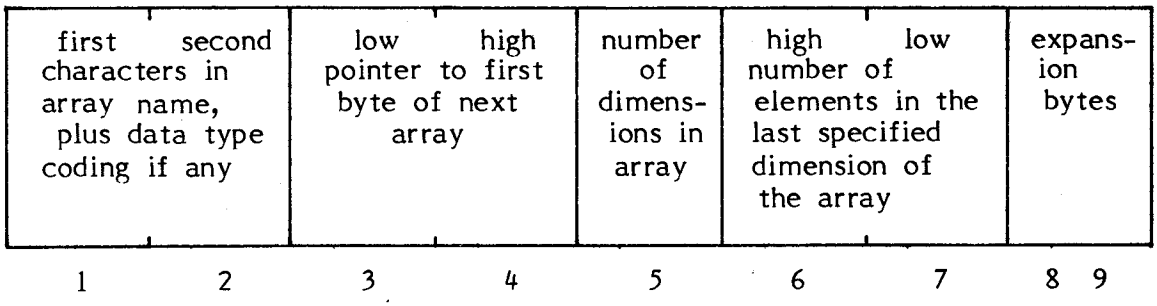
Since the number of characters in the string is stored as a single byte it is not possible to have a character string longer than 255 characters. This should be considered when adding two string variables together where both are fairly long. Though the area at the top of memory is allocated for the storage of strings, not all string variables are stored there. Thus all strings defined within the program are retrieved, when required from the program text area. This is done by having the variable address pointers point to the location in Basic text rather than the top of memory. What is stored at the top of memory are calculated string variables. The area of memory occupied by these strings can be determined by looking at the contents of locations 48

and 49 (in old ROMs 130 and 131) this is the start address of the string area, and 50 and 51 (in old ROMs 132 and 133)which is the end address.

The three data types encountered as simple single value variables can also be stored as multiple value or array variables. Whereas simple variables of whatever data type all occupy the same amount of memory for each variable, the memory requirement for an array is different for each type of data. An array is stored as; an array header plus a set of elements each roughly corresponding to a simple variable. The array header contains the array name, the number of dimensions in the array, the number of elements in each dimension together with a pointer to the start of the next array. Array header are the same for all data types. As with simple variables the array data type is coded into the array name. In a floating point array both characters are the normal ASCII code, in an integer 128 is added to the ASCII value of both characters, and in a character array 128 is added to the ASCII value of the second character only. The general format of an array is:



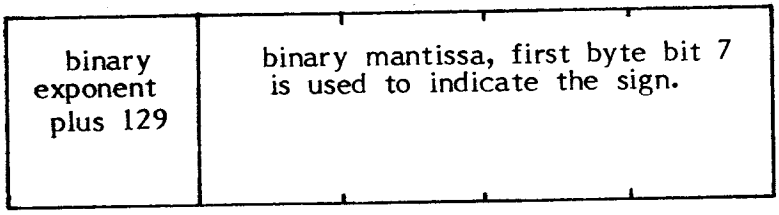
Here N is used to designate the last element in an array and corresponds to the value used in the DIM statement at the beginning of the program when the array was initialised. The array header for whatever data type has the format:



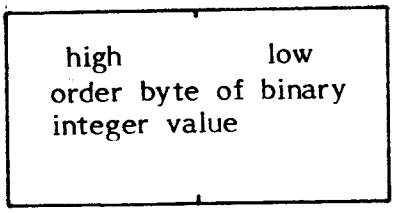
In a one dimensional array the array header occupies seven bytes, but if two dimensions are specified then an extra two bytes are required to specify the number of elements in that dimension, making the header nine bytes long. Similarly if there are three dimensions it would be eleven bytes long. In a two dimensional array set up by DIM D(A,B) the number of elements in B is stored in

bytes 6 and 7 of the header, the number of elements in A is stored in bytes 8 and 9. The format for each element in an array is identical since all elements are of the same data type, though the format is different for each data type:

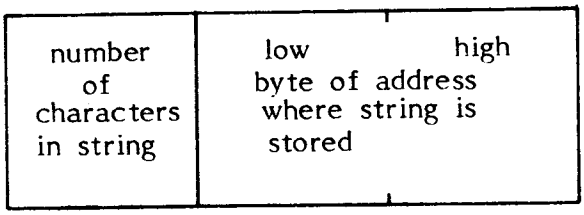
FLOATING POINT ARRAY ELEMENT



INTEGER ARRAY ELEMENT



CHARACTER ARRAY ELEMENT



NOTE: a negative integer whether in an array or a simple variable is stored as a twos complement number, thus no sign bit is used and negative integers can not exceed 32768.

An annoying limitation of array variables in old ROM machines is the maximum of 255 elements in an array (this has been overcome in the new ROM machines). One fairly simple way of overcoming this problem is to construct ones own arrays using the PEEK and POKE commands, then the only restriction is the amount of free memory available in the system. Since data is to be stored without using the Basic arrays or variables an area of memory must be set aside exclusively for the storage of the new arrays. The way to do this is to lower the top of memory pointers until it is just above the maximum area required for program storage, strings and variables. This can be calculated by using the FRE command to determine the program size and adding to it

the amount of memory required to store variables and strings. The space required for storage of simple variables is obtained by counting the number of variables and multiplying by seven. If array variables are used then the memory requirements depend on the data type and number of dimensions, but can be calculated as the header, plus the number of elements, times the number of bytes in each element. The amount of memory required for string storage is obtained by counting the maximum number of characters which will be stored as strings,(only calculated or input strings need be counted). Having obtained a figure for the maximum amount of memory required to run the program this can be subtracted from the total user memory area to give the amount of free memory.

To use this free memory area, one must first calculate the number of bytes required to store each variable in the proposed array. Great care must be taken with this if the maximum amount of data is to be stored in a given area of memory. The method used to store the data will also affect the speed with which data can be accessed from the array. If linear search techniques are used this could slow down a program considerably. If the array consists of a table of character strings then one of two methods could be used. The choice depends on whether access speed is more important than amount of data stored per K byte.

The first method is to store character strings of any length, with a maximum size of say 255 bytes, the first byte of each string indicating the length of that string. Searching through a file stored using this method requires a slow linear search, since the contents of the first byte of each string is used to point to the start of the next string. The second method is to allocate a fixed amount of memory space to each string, the number of characters depending on what is being stored, however all elements in the array must have the same space allocated to them. The search procedure here is very easy since if we want the contents of element 14 it is located at an address which can be calculated by adding the array starting address to 14 times the number of bytes in each element. The only problem with this method is that character strings shorter than the maximum will leave unused spaces in memory and if longer then it is impossible to store the extra charaters.

Elements in a numerical array can be stored as either binary numbers or as ASCII values the method used depending on the maximum size of the numbers stored. Whichever method is used it is preferable to have all elements in the array the same size. If numbers are stored as binary then a three byte element can be used to store numbers in the range +65535 to -65535 the first byte being used to store the sign. To find element number N in an array one simply calculates its starting

position by adding the array starting address to N times the number of bytes in each element. Using this method one can create a thousand element array for numeric variables in the range +-65536 in 3K of memory or if all values are positive then it could be stored in just 2K.

Programs involving extensive string manipulation can suffer from seemingly inexplicable and often lengthly pauses in their operation. This is caused by an operating system function known as garbage collection. Every time a character string is input or calculated it is stored at the bottom of the character string storage area. If A$ is input at the beginning of the program, and then at the end another A$ string is input, the second input is not stored on top of the first but at the end of the string storage space, leaving the first string still stored in memory. Obviously if the program involves a fair amount of string manipulation the entire free memory space will become full of string storage, a large proportion of which will be "garbage" i.e. strings no longer required. To avoid running out of memory the system must perform at this point a "garbage collection" routine. Garbage collection reclaims all the unused memory and compacts the string storage at the top of memory. This subroutine which is located at D400 to D496(D404 to D5C3 in old ROMs) is lengthy and time consuming especially in large programs and the main reason why such programs execute at a much slower rate than small programs. One can force garbage collection to take place by performing the command FRE (O), which calculates the amount of free memory space, this is useful if you don't want a real time program interrupted by the garbage collection process.

When the command RUN is typed on the screen followed by a carriage return, the operating system interprets this as a direct command. It then searches through the list of reserved words to find the address of the subroutine to perform the command. The RUN subroutine is located at address C785 (C775 in old ROMs)its first function is to set all the pointers to the start of the program, abort all active I/0 channels and restore all subroutine and data pointers. Having done this, the first line of the Basic program is fetched using a subroutine located in page zero of memory. The command is executed, and the next line fetched, with the line fetch subroutine checking for spaces and more than one command on a line.

The line fetch subroutine in page zero is of great interest, since it opens up the possibility of adding additional commands to Basic. For this reason it is worth looking at the subroutine closely, it is loaded into memory from locations 112 to 117 (in old ROMs 194 to 199) during system initialisation. The reason why this subroutine is relocated from ROM to RAM is that it requires a variable load address. This points to the

current byte of Basic program text being accessed. The variable load address or pointer to source text is stored in locations 119 and 120 (in old ROMs 201 and 202). The first function of the subroutine is to increment this pointer to point to the next location, which is then read and stored in the processors accumulator. The remainder of the subroutine checks to see if the character obtained is either a colon, indicating the end of a statement, or a space, if a space then the next character is obtained. The subroutine is as follows, new ROM version:

```
0070   CHAR   E6 77      INC Z  $77   :increment character
                                       pointer low byte
              D0 02      BNE $02      :test if low byte=255
                                       if true then
              E6 78      INC Z  $78   :increment character
                                       pointer high byte
       GET    AD ** **   LDA ****     :get character from
                                       address in 119-120
              C9 3A      CMP IMM $3A  :is character a colon
              B0 0A      BCS END      :if so then End
              C9 20      CMP IMM $20  :is character a space
              F0 EF      BEQ CHAR     :if so goto CHAR
              38         SEC
              E9 30      SBC $30
              38         SEC
              E9 D0      SBC $D0
       END    60         RTS          :return to main program
```

(the asterisks are used to show that the contents of bytes 201 and 202 are variable).

By inserting extra code into this subroutine,(this is done by replacing the first six bytes with a couple of jumps to user written code) each Basic command can be intercepted before it is performed. The first subroutine would be the main block of new code, performing whatever function one wants to add to the PET commands. The second subroutine consists of the six bytes replaced by the two JSR instructions. Thus if the new program starts at location 7AOO hex than the following six bytes would be inserted into the CHARGOT area:

```
0070   CHARGOT   20 00 1A   JSR  1A00   :main subroutine
                 20 00 1F   JSR  1F00   :update pointer
                                         subroutine
       GET       AD ** **   LDA  ****
                    .          .
                    .          .


1F00   POINTER   E6 77      INC Z $77   :increment 119
```

```
        D0 02     BNE $02     :if low byte =255 then
        E6 78     INC Z $78   :increment 120
END     60        RTS         :return to main program
```

An easy way of detecting new commands is to precede them by a particular character, eg. an asterisk. Then use a small subroutine to detect if the first character in a command is an asterisk. If so, then the command is executed by the new software rather then the existing interpreter. A vector plotting command could be added, to plot line vectors on the screen using double density graphics (see the program for this in the section on the video display). A command like *PLOT X1,Y1,X2,Y2 could be used where X and Y are co-ordinate values for the end points of the line. The range of commands is very large, including functions like the example just given, also commands governing the operation of peripherals such as A/D converters, or disk units. The ability to intercept each command before it is executed need not be applied to adding extra commands to Basic. It can also be used to monitor the execution of a program, allowing one to construct a powerful diagnostic aid known as a trace program, which slows down the running of a Basic program and displays each line on the screen as it is executed. The following programs perform this function, since they are fairly lengthy machine code programs I will not give the full source text, only the loader written in Basic.

The first commands set the top of memory pointers so that trace will not be erased by any Basic variables or strings since it resides above the top of memory pointer. The trace program should be loaded first before entering or loading the program which is to be tested using trace. Once trace and the program to be examined are loaded, then trace can be activated. In the first program which is for old ROM machines trace is enabled by the command--- SYS (7876), this inserts the new code into the CHARGOT subroutine as explained above. The second program is a version of trace for new ROM machines. In this version to allow machines of different sizes to run trace the SYS locations are calculated by the Basic loader and should be noted prior to running. Having activated trace the program to be examined can by run by typing RUN in the normal manner, the program will then be executed. Each line being executed is displayed in two lines of reverse field background at the top of the display at the rate of about one line every second. The rate of program execution can be speeded up in the old ROM version by pressing the shift key, the new ROM version requires a speed flag to be reset using POKE. Program execution can be stopped in the normal manner by pressing the stop key.

It should be noted that when trace has been initialised it affects the operation of the cassettes

and the I/0 thereby rendering it impossible to either load or save a program. To overcome this problem a disable subroutine has been built into trace. This subroutine returns the CHARGOT subroutine area to its normal state and can be called in the old ROM version by a --- SYS(7861).

```
 1 REM TRACE FOR OLD ROM MACHINES
 10 FORQ=7853TO8191
 20 READA
 30 POKEQ,A
 40 NEXTQ
 50 END
 100 DATA 162, 5, 189, 181, 224, 149, 194, 202, 16, 248, 169
 110 DATA 239, 133, 210, 96, 169, 172, 133, 134, 169, 30
 120 DATA 133, 135, 169, 255, 133, 124, 160, 0, 162, 3
 130 DATA 134, 125, 162, 3, 32, 239, 30, 208, 249, 202
 140 DATA 208, 248, 32, 239, 30, 32, 239, 30, 162, 5
 150 DATA 189, 249, 31, 149, 194, 202, 16, 248, 169, 242
 160 DATA 133, 210, 76, 106, 197, 230, 124, 208, 2, 230
 170 DATA 125, 177, 124, 96, 230, 201, 208, 2, 230, 202
 180 DATA 96, 32, 197, 0, 8, 72, 133, 79, 138, 72
 190 DATA 152, 72, 166, 137, 165, 136, 197, 77, 208, 4
 200 DATA 228, 78, 240, 107, 133, 77, 133, 82, 134, 78
 210 DATA 134, 83, 173, 4, 2, 208, 14, 169, 3, 133
 220 DATA 74, 202, 208, 253, 136, 208, 250, 198, 74, 16
 230 DATA 246, 32, 201, 31, 169, 160, 160, 80, 153, 255
 240 DATA 127, 136, 208, 250, 132, 76, 132, 84, 132, 85
 250 DATA 132, 86, 120, 248, 160, 15, 6, 82, 38, 83
 260 DATA 162, 253, 181, 87, 117, 87, 149, 87, 232, 48
 270 DATA 247, 136, 16, 238, 216, 88, 162, 2, 169, 48
 280 DATA 133, 89, 134, 88, 181, 84, 72, 74, 74, 74
 290 DATA 74, 32, 211, 31, 104, 41, 15, 32, 211, 31
 300 DATA 166, 88, 202, 16, 233, 32, 217, 31, 32, 217
 310 DATA 31, 165, 75, 197, 201, 240, 55, 165, 79, 208
 320 DATA 4, 133, 77, 240, 47, 16, 42, 201, 255, 208
 330 DATA 8, 169, 94, 32, 225, 31, 24, 144, 33, 41
 340 DATA 127, 170, 160, 0, 185, 145, 192, 48, 3, 200
 350 DATA 208, 248, 200, 202, 16, 244, 185, 145, 192, 48
 360 DATA 6, 32, 223, 31, 200, 208, 245, 41, 127, 32
 370 DATA 223, 31, 165, 201, 133, 75, 104, 168, 104, 170
 380 DATA 104, 40, 96, 168, 173, 64, 232, 41, 32, 208
 390 DATA 249, 152, 96, 9, 48, 197, 89, 208, 4, 169
 400 DATA 32, 208, 2, 198, 89, 41, 63, 9, 128, 132
 410 DATA 81, 32, 201, 31, 164, 76, 153, 0, 128, 192
 420 DATA 79, 208, 2, 160, 7, 200, 132, 76, 164, 81
 430 DATA 96, 76, 255, 30, 32, 248, 30, 36, 239, 255
READY.
```

```
1 REM TRACE FOR NEW ROM MACHINES
10 E=52
15 D=2
100 DATA-342,162,5,189,249,224,149,112,202,16,248
110 DATA169,239,133,128,96,173,-342,133,52,173,-341
120 DATA133,53,169,255,133,42,160,0,162,3,134,43
130 DATA162,3,32,-271,208,249,202,208,248,32,-271
140 DATA32,-271,76,121,197,162,5,189,-6,149,112,202
150 DATA16,248,169,242,133,128,96,230,42,208,2,230
160 DATA43,177,42,96,230,119,208,2,230,120,96,32
170 DATA115,0,8,72,133,195,138,72,152,72,166,55,165
180 DATA54,197,253,208,4,228,254,240,106,133,253
190 DATA133,35,134,254,134,36,165,152,208,14,169
200 DATA3,133,107,202,208,253,136,208,250,198,107
210 DATA208,246,32,-54,169,160,160,80,153,255,127
220 DATA136,208,250,132,182,132,37,132,38,132,39
230 DATA120,248,160,15,6,35,38,36,162,253,181,40
240 DATA117,40,149,40,232,48,247,136,16,238,216
250 DATA88,162,2,169,48,133,103,134,102,181,37,72
260 DATA74,74,74,74,32,-44,104,41,15,32,-44,166
270 DATA102,202,16,233,32,-38,32,-38,165,184,197
280 DATA119,240,55,165,195,208,4,133,253,240,47
290 DATA16,42,201,255,208,8,169,105,32,-30,24,114
300 DATA33,41,127,170,160,0,185,145,192,48,3,200
310 DATA208,248,200,202,16,244,185,145,192,48,6
320 DATA32,-32,200,208,245,41,127,32,-32,165,119
330 DATA133,184,104,168,104,170,104,40,96,168,173
340 DATA64,232,41,32,208,249,152,96,9,48,197,103
350 DATA208,4,169,32,208,2,198,103,41,63,9,128
360 DATA132,106,32,-54,164,182,153,0,128,192,195
370 DATA208,2,160,7,200,132,182,164,106,96,76
380 DATA-255,32,-262
400 S2=PEEK(E)+PEEK(E+1)*256:S1=S2+D-344
410 FORJ=S1TOS2-1
420 READX:IFX>OORX=OTHENGOTO450
430 Y=X+S2:X=INT(Y/256):Z=Y-X*256
440 POKEJ,Z:J=J+1
450 POKEJ,X
460 NEXTJ
500 PRINT"INITIALISE WITH     SYS(";S1+17")"
510 PRINT"ENABLE WITH         SYS(";S1+56")"
520 PRINT"DISABLE WITH        SYS(";S1+2")"
530 PRINT"CHANGE SPEED WITH   POKE";S1+125-D",X"
600 END
READY.
```

THE USER PORT 4

An understanding of the functioning and programming of the user port and the 6522 VIA is vital for anyone wishing to use the PET to control or communicate with an external device. The user port is the central edge connector coming from the main PET logic board at the rear of the machine. It has 24 edge connections 12 on the top and 12 on the bottom, with a .156 inch spacing between the centre of each contact. We can divide the 24 contacts into two distinct groups, the 12 top contacts and 12 bottom contacts. The top 12 connections are primarily intended for use when servicing the PET, the bottom 12 lines make up the parallel user port. A brief description of each contact is shown in Figure 4.1, the top connections are labeled 1 - 12 and the bottom A - N.

Connections 1 - 12 and their uses.

The top connections are of little use to the average user and in general should be treated with caution. However, on the old 8K static RAM machines they were designed as part of the internal diagnostics. For this purpose a special connector is used to jumper some of the top contacts to the bottom contacts. With this connector in place the PET when powered up instead of jumping to the BASIC routines jumps to the diagnostic routines contained in the PETs ROM. The diagnostic routine checks the RAM, parity of the ROM, keyboard scanning, TV display (making sure all bits turn on and off at all locations on the screen), Read/Write of both cassette ports, user and IEEE port. If all functions of the PET main logic board are working correctly on completion of the test the red LED on the board will turn on. In fact two diagnostic connectors are required to do this, one on the user port, the other on the keyboard connector in place of the keyboard cable. If you wish to run the diagnostics you will have to make up your own connectors with the following connections wired together:

1	Ground	Digital ground
2	T.V.video	Video output used for external display used in diagnostic routine for verifying the video circuit to the display board.
3	IEEE-SRQ	Direct connection to the SRQ signal on the IEEE-488 port. It is used in verifying operation of the SRQ in the diagnostic routine.
4	IEEE-EOI	Direct connection to the EOI signal on the IEEE-488 port. It is used in verifying operation of the EOI in the dignostic routine.
5	Diagnostic Sense	When this pin is held low during power up the PET software jumps to the diagnostic routine, rather than the BASIC routine
6	Tape #2 READ	Used with the diagnostic routine to verify cassette tape #2 read function.
7	Tape Write	Used with the diagnostic routine to verify operation of the WRITE function of both cassette ports.
	Tape #1 READ	Used with the diagnostic routine to verify cassette tape #1 function.
9	T.V. Vertical	T.V. vertical sync signal verified in diagnostic. May be used for external T.V. display.
10	T.V. Horizontal	T.V. horizontal signal vertified in diagnostic may be used for T.V. display.
11,12	GND	Digital ground
A	GND	Digital ground
B	CA1	Standard edge sensitive input of 6522VIA.
C	PA0	Input/output lines to peripherals, and can be programed independently of each other for input or output.
D	PA1	
E	PA2	
F	PA3	
H	PA4	
J	PA5	
K	PA6	
L	PA7	
M	CB2	Special I/0 pin of VIA.
N	GND	Digital ground

Fig 4.1 User Port Connections

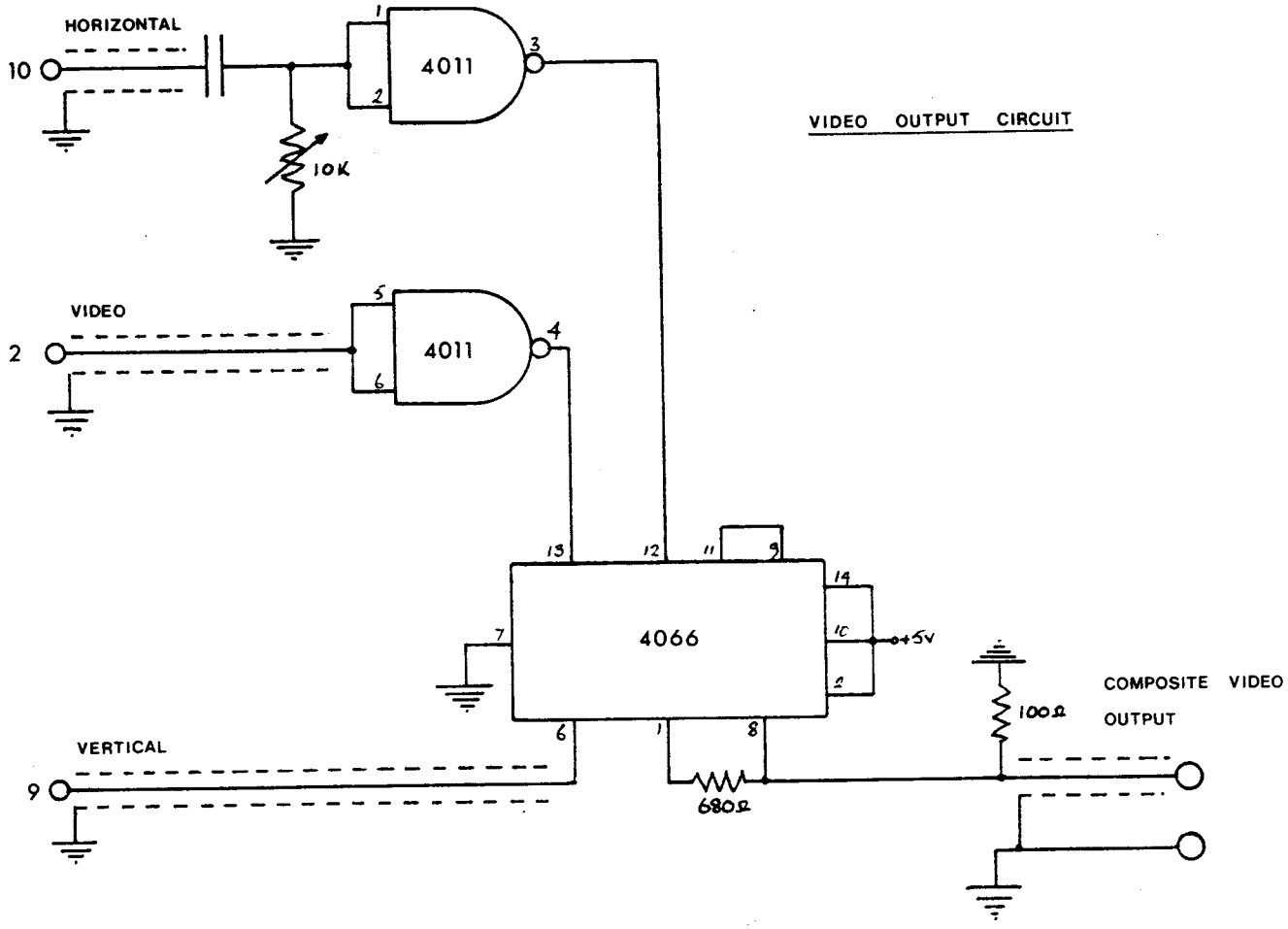
User port connector - 2-B, 3-C, 4-D, 5-11, 6-7-8, 9-K, 10-L

Keyboard connector - 1-9-17, 2-10-18, 3-11, 4-12, 5-13, 6-14, 7-15, 8-16, connector key in position 19.

When power is applied to the PET the screen will initially be cleared and the red LED will be off. The diagnostic will begin by testing the screen while doing this a small white square will sweep across all the locations on the screen. On the cursor reaching the bottom right of the screen the display will be filled with a full character test pattern. This should be checked visually to make sure all the characters are present and no bits are flickering on the screen. The red LED should be lit indicating that the main logic board has passed the diagnostic test. If a fault is present it can be tracked down with a set of diagnostic programs loaded into the PET from tape, of course no diagnostic routines will work if the processor is not functioning or there is no power. All new machines require diagnostic programs to be loaded from either tape or disk.

Although primarily intended for use by the diagnostics the top connections of the user port can be used for other purposes. One of the most useful, with particular applications in schools, is the ability to use three of these lines to drive an external large screen TV monitor. These three lines provide the user with the vertical sync signal on output 9, the horizontal drive on 10 and the video output on line 2. To drive a standard TV monitor (not a domestic TV set) these signals must be combined to give a single composite video output which can be connected directly to the monitor input. The circuit to do this is shown in Figure 4.2, it will require a 5 volt power supply which can come from a battery or from pin 2 on the 2nd. cassette connector. You may encounter problems with the horizontal hold but this can usually be cured by adjusting the value of R1. Other problems may occur as a result of using very cheap monitors or converted TV sets. If you intend building this circuit then the actual layout of components is not critical and it is most easily constructed on a piece of Veroboard.

The only other lines of interest on the top surface of the user port connector, are lines 6, 7 and 8, these are all associated with the operation of the two cassettes. Line 6 is the read input from cassette #2 and line 8 is the read input from cassette #1, line 7 is the common write output to both cassette decks(dynamic machines only). These lines are of interest to the user for several reasons, line 7 could be used as an extra I/O line. The most interesting application lies in using these lines to allow two or more PETs to communicate



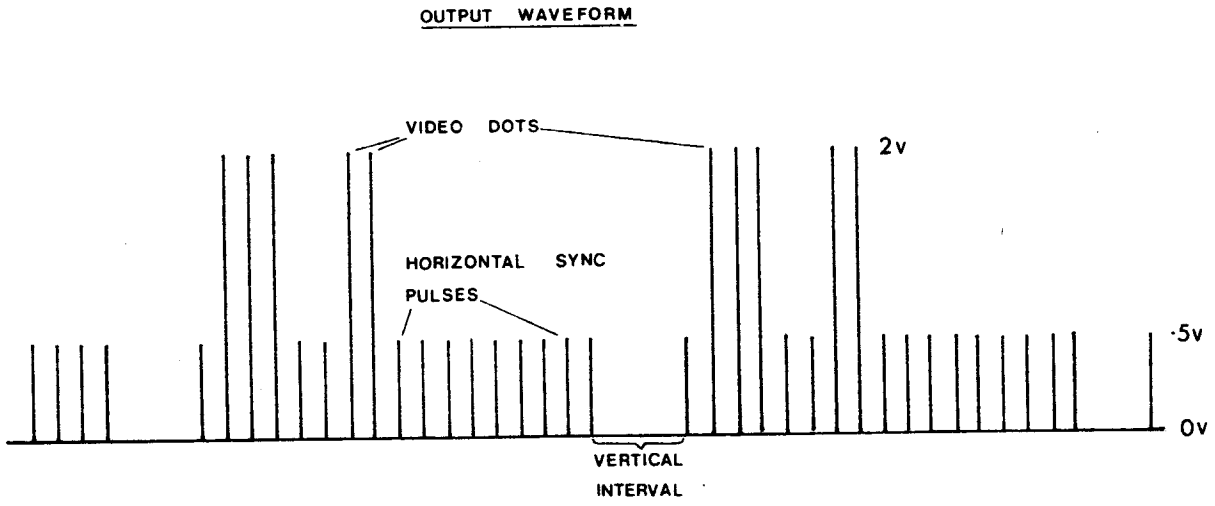


Fig 4.2 Video output circuit

with each other. This would allow one master PET to control a number of slave machines a situation which would find great use in education. "communicate" meaning that data and programs can be transferred from one machine to another. This is done by connecting the write output of one machine to the read input of another, and vice versa to give bi-directional communication. At the time of writing several people are experimenting with this idea, though the results look promising no working system has yet been constructed.

The parallel user port, connection A - N.

The bottom 12 connections comprise the user port proper, and are of interest to everyone wishing to use the PET to control external devices. As seen from Figure 4.1 these 12 lines consist of two ground lines, two handshaking lines and eight input/output lines. Since the I/0 lines are under full program control they can be configured as any combination of inputs or outputs. This means that the user port should not be considered as an eight line data bus like the IEEE, but rather as a set of eight independent I/0 lines. Examples of the kind of devices which could be connected to the user port are: lamps - switches and other on/off sensing devices - motors - anolog to digital or digital to analog converters. Some of these devices could be controlled with programs written in Basic, but the majority would require the control program to be written in machine code since Basic is not fast enough for most applications. To write programs either in Basic or machine code for controlling devices through the user port the programmer must have a thorough understanding of the functioning of the I/0 chip from which the user port lines originate.

The 6522 Versatile Interface Adapter

The lines on the bottom side of the user port originate from a 6522 VIA or Versatile Interface Adapter chip, located in system memory between addresses 59456 and 59471. A block diagram of the 6522 is shown in Figure 4.3, it is a very complex chip with sixteen different addressable registers. Each bit within these registers has a specific function, either as an input, an output or to control the operation of the 6522. A memory map of the addressable registers is shown in Figure 4.4, the registers are of six basic types; I/0, data direction, peripheral control, shift register, timers and timer control registers.

The diagram in Figure 4.3 can be divided into two, on the left are the connections to the processor, the processor interface. On the right the outputs of the 6522, or the peripheral interface. The main components

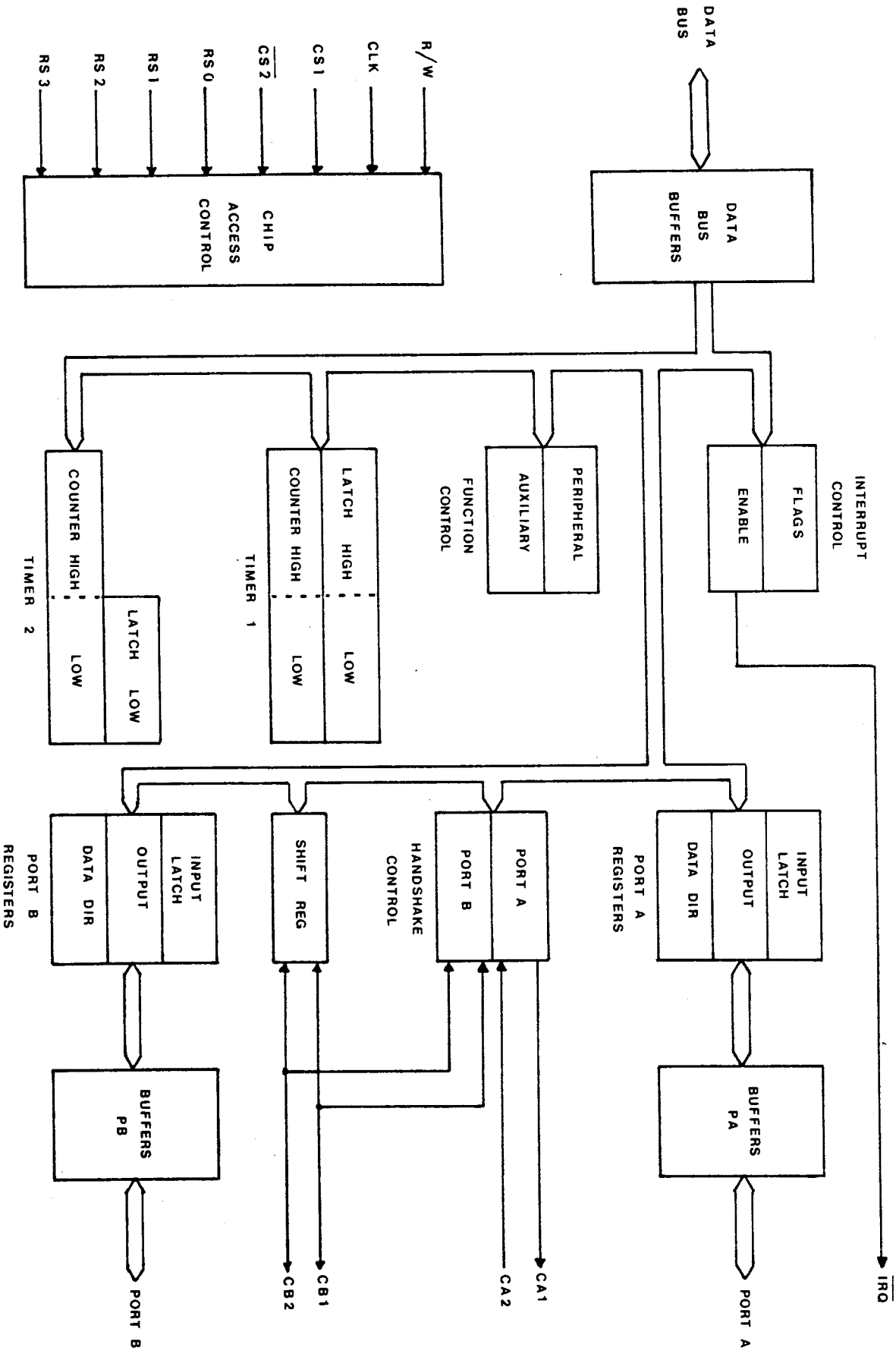


Fig 4.3 6522 Block Diagram

of the processor interface are the eight bi-directional data lines. These are connected directly to the processor data bus and are used to transfer data between the VIA and the processor. As with any memory, the processor treats the 6522 as a sixteen byte block of memory, the direction of data transfer is controlled by the R/W line, the exact timing of a transfer being controlled by the o2 clock line. The individual registers are addressed by the register select lines connected to the bottom address lines AO - A3. The exact location of the 6522 within memory space is determined by decoding some of the address lines and connecting these to the chip select inputs. The registers of the 6522 will only be accessed if chip select CS1 is high and CS2 low. CS1 is connected via an AND gate to address lines 11 and 6, and CS2 to memory block select line E. As with all the I/0 chips the 6522 can generate a processor interrupt by pulling the IRQ line low. This occurs whenever an internal interrupt flag is set as a result of an input on one of the peripheral control lines.

The peripheral interface consists of two eight line I/0 ports, port A and port B, together with their associated control lines. The eight lines of port A can be individually programmed to act as either inputs or outputs under control of the data direction register. Input data is latched onto an internal register under control of the CAI line and the polarity of any outputs is controlled by the contents of the output register. The two port A control lines CA1 and CA2 act either as interrupt inputs or handshake outputs. In the interrupt mode each line controls an internal interrupt flag, CA1 also controls the latching of input data on port A.

The eight I/0 lines of port A plus control line CA1 go to the user port. Control line CA2 is connected to the character generator and controls the lower case/graphics mode. Port B is identical to port A except that the polarity of an output on line PB7 can be controlled by the interval timers and the second internal counter can be programmed to count pulses input on line PB6. The peripheral B control lines CB1 and CB2 perform the same functions as CA1 and CA2 but in addition can act as a serial port under control of the shift register. The lines of port B perform a wide range of system and I/0 functions thus:

PB0 - NDAC input from IEEE port
PB1 - NRFD output to IEEE port
PB2 - ATN output to IEEE port
PB3 - cassette write output
PB4 - cassette #2 motor control
PB5 - Video on control
PB6 - NRFD input from IEEE port
PB7 - DAV input from IEEE port

Control line CB1 is the read input for cassette 2 while CB2 is the second control line on the user port.

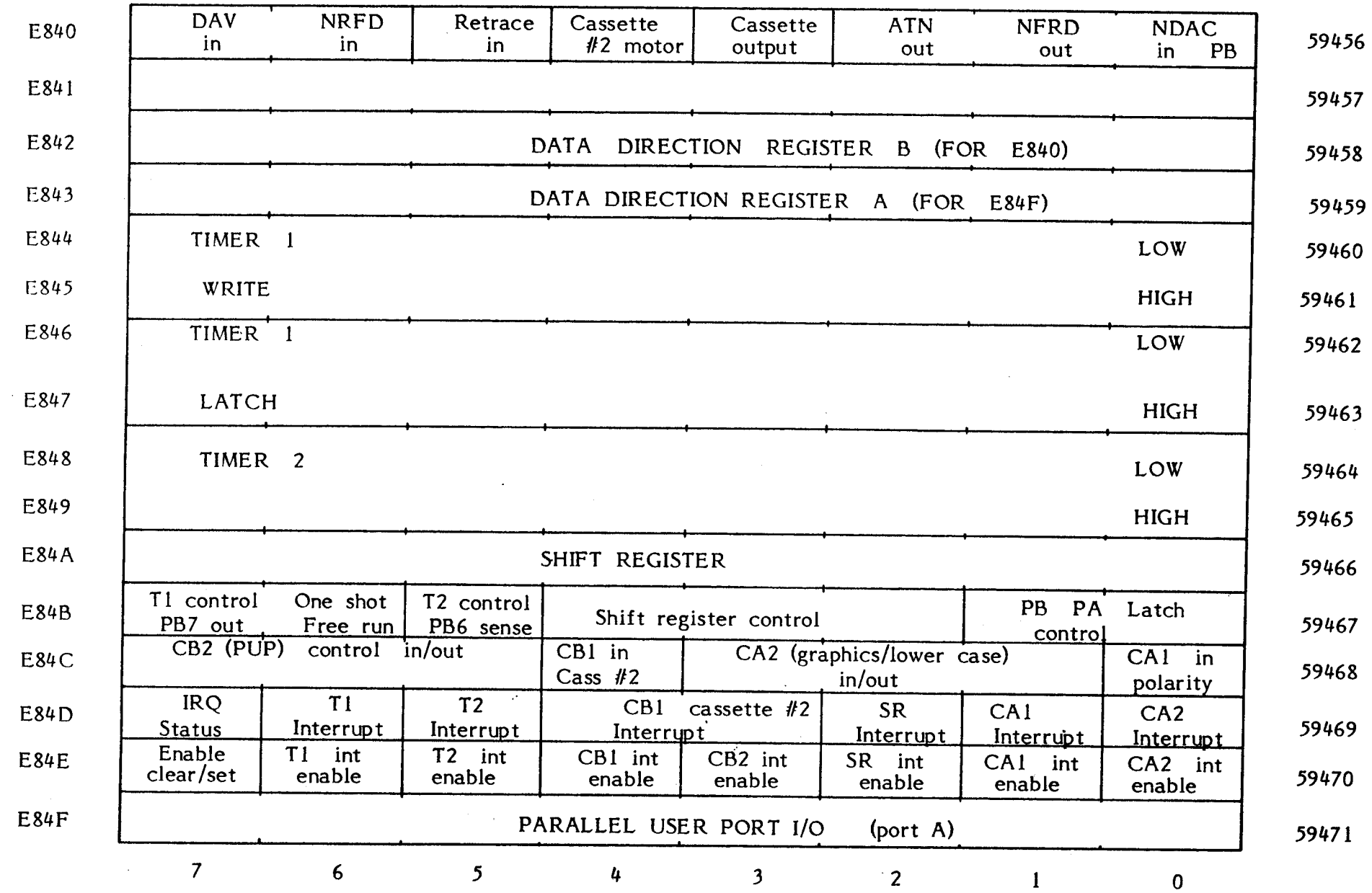
Using the Parallel I/0 ports..

Three registers are directly associated with each of the eight line peripheral I/0 ports, they are the data direction register, input register and output register. The data direction registers DDRA, DDRB, are used to specify whether a particular line acts as an input or output. Each bit in the DDR corresponds to a line in the I/0 port, if the contents of that bit is a 0 then the corresponding line is an input, if the contents is a 1 then it will be an output. If DDRA was loaded with 00001111 by a POKE 59459,15 then lines PA0 - PA3 on the user port would be configured as outputs and PA4 - PA7 as inputs. Each line on the port is connected to a corresponding bit in both the output and input registers, each being enabled or disabled by the output of the data direction register. If an I/O line is programmed as an output by the contents of the DDR then the voltage on that line is controlled by the corresponding bit in the output register, 0 causes the line to go high, a 1 low. Any data written into output register bits corresponding to lines programmed as inputs will have no effect on those lines. As an example four lines PA0 - PA3 of the user port are configured as outputs with POKE 59459,15 then:-

	POKE 59471,255	PA0-3 go high, PA4-7 are unaffected
or	POKE 59471,0	PA0-3 go low, PA4-7 are unaffected
or	POKE 59471,3	PA0-1 are high, PA2-3 are low, PA4-7 are unaffected

In this manner any one or more lines on either of the peripheral ports can be configured as an output by a program. Also under program control the voltage on output lines can be set either high or low. This allows the programmer colossal flexibility in the use of I/0, in one instant a line can be configured as an output in the next the same line can be an input.

If a line is configured as an input by the data direction register then the corresponding bit in the input register will reflect the voltage level on that line. Reading the input port will transfer the contents of the input register onto the processor data bus. Since data is being input to the VIA asynchronously an input may be changing as the processor is reading it, the resulting input being erroneous. Synchronisation is established by using handshaking lines. CA1 acts not only as an interrupt input but at the same time latches


E840
DAV in
NRFD in
Retrace in
Cassette #2 motor
Cassette output
ATN out
NFRD out
NDAC in PB
59456
E841
59457
E842
DATA DIRECTION REGISTER B (FOR E840)
59458
E843
DATA DIRECTION REGISTER A (FOR E84F)
59459
E844
TIMER 1
LOW
59460
E845
WRITE
HIGH
59461
E846
TIMER 1
LOW
59462
E847
LATCH
HIGH
59463
E848
TIMER 2
LOW
59464
E849
HIGH
59465
E84A
SHIFT REGISTER
59466
E84B
T1 control PB7 out
One shot Free run
T2 control PB6 sense
Shift register control
PB PA Latch control
59467
E84C
CB2 (PUP) control in/out
CB1 in Cass #2
CA2 (graphics/lower case) in/out
CA1 in polarity
59468
E84D
IRQ Status
T1 Interrupt
T2 Interrupt
CB1 cassette #2 Interrupt
SR Interrupt
CA1 Interrupt
CA2 Interrupt
59469
E84E
Enable clear/set
T1 int enable
T2 int enable
CB1 int enable
CB2 int enable
SR int enable
CA1 int enable
CA2 int enable
59470
E84F
PARALLEL USER PORT I/O (port A)
59471
7
6
5
4
3
2
1
0

VIA (6522)

any input data into the input register. The peripheral ports can be in either a latched or unlatched mode, depending on the state of the latch enable flags in the auxiliary control register. In the latched mode, the enable flag is 0. Data present on the peripheral port input lines will be latched into the input register when the CA1 or CB1 interrupt flag is set by an active transition from high to low on the CA1 or CB1 line. As long as the CA1 or CB1 interrupt flag is set, data on the peripheral input lines can change without affecting data in the latched input register. Data can also be latched into the register by setting the CA1 or CB1 interrupt flag from a program, similarly program instructions can be used to clear the interrupt flag.

When using a handshaking line to control the latching of data into the input register from an external device, it is important to make sure that data on the input lines has stabilised prior to an active transition on the hankshake line. The input of data on ports A and B is identical except that whereas in port B the state of the output lines is always reflected into the corresponding bit of the input register, in port A this may not always be the case.

Inputting data from the user port is considerably more complex than outputting, since it can be done in two ways. Firstly by reading the input port, secondly by an interrupt service routine. The method employed depends primarily on the frequency that the input will be read by the program, also whether the programmer can allow the processor to wait for an input. If all the program requires is the current state of one or more input lines where the exact timing of that input is not important, then simply reading the input will suffice. If however a series of inputs occurring at a particular time are to be recorded, then the computer must stop and repeatedly test for an input. When one occurs it is stored in the relavent location, the processor then returns to look for another input. Two methods can be employed to do this, if processor time is not important then one simply repeatedly scans the input. On each scan the contents of bit 1 of the interrupt flag register is tested to see if any data has been latched into the input register by a transition on CA1. If it has then the input register is read, otherwise the processor repeats the test loop waiting until an input occurs. Such a program could be written in either Basic or machine code the choice depending on the frequency of the inputs. In Basic the maximum frequency is about 40Hz, in machine code 50KHz. It is often not practical to make the processor wait for an input, to overcome this the input scanning routine can be made part of the interrupt sequence occuring 60 times a second in the PET. Such a machine code program incorporated in the interrupt software will search for an input every

sixtyth of a second independent of any program or use to which the machine is being put (with the exception of Loading and Saving programs or data, and communication on the IEEE port).

The simplest form of input is to read the contents of the input register whenever the contents are required by the program. It may be necessary at a particular point in a program to know if a switch connected to one of the input lines is 'on' or 'off'. Where 'on' means that the line is at a high logic level (+5 volts) and 'off' is a low level (0 volts). Since the state of the switch changes infrequently there is no need to latch the data on the input line into the input register with the aid of handshaking line CA1. A program to test the state of a switch connected to line 7 of the input port could be like this:

```
100 POKE 59459,127
105 REM SET DDRA:  PA7 IS AN INPUT, REST OUTPUTS
110 A=999
120 K=PEEK(59471): REM READ INPUT REGISTER
130 C=128 AND K:REM MASK OFF BITS 0 TO 6
140 IF A=C THEN 160
145 REM TEST FOR STABILITY OF INPUT DATA BY LOOKING
146 REM AT THE INPUT TWICE AND CHECKING FOR A CHANGE
150 A=C:GOTO120
160 IFC=128 THEN PRINT "SWITCH ON":GOTO180
165 REM PRINT RESULT
170 PRINT"SWITCH OFF"
180 END
```

Note that because the values of bits 0 to 6 of the input register are unknown these must be masked off by ANDing the input with binary 10000000 - decimal 128. If the result of this logical operation is 128 then bit 7 of the input register is set and therefore the switch is on, if not then by default the switch is off. The reason the input is read twice is to make sure that the state of the input was not changing at the same instant it was being read. Rather than just reading the current state of an input the programmer may want the computer to wait until a specific input occured like a switch being turned on. This could be done in several ways all of which involve the processor repeatedly reading the input register and testing for the required input. Since the processor is waiting for an input there is no need to latch the input into the input register with a pulse on the CA1 line, unless the input is of very short duration and likely to be missed. If as in the last example a switch is connected to line 7 of the user port which is defined by the data direction register as an input then either one of the following two lines in Basic will cause the computer to wait for an input.

```
110 IF (PEEK (59471) AND 128) THEN 110
        or
100 WAIT 59471,128
```

In the first example the switch is normally open and the voltage on the input line floats to a high level. This line of program causes the processor to halt until the switch is closed and the input line connected to ground. In the second program line the reverse is true, the switch is normally closed and the input connected to ground. This line causes the processor to wait until the switch is opened. Both lines of program will scan the input port looking for the correct input on one or more lines about a hundred times a second. If the data expected by the program on the input lines is present for less than one fiftieth of a second the inputs must either be latched or the scanning program written in machine code. The WAIT statement should be used with care since the processor will wait until the contents of a specified memory location contains a particular value. One cannot break out of the Wait statement by pressing the Stop key on the keyboard, any mistakes in coding or failure to input the right value will cause the machine to crash.

The methods of inputting data looked at so far would be used with sensor devices connected to the computer. In these applications it is the state of the line, i.e. either logic high or logic low at a particular time which is of interest, rather than the changing of the state of that line with respect to time. Sampling the input data at regular intervals can be done by using a timed program loop to repeatedly read the input register and store each input in a table. As an example: an eight bit analog to digital converter connected to the user port. A record of the voltage is to be kept sampled once every second with a maximum of 100 samples. Each sample is stored in a dimensioned array, the timing of each sampling is controlled by using the jiffy clock (variable TI) on the PET.

```
10 DIMA(100)
100 FORQ=1TO100
110 T=TI
120 IFTI<T+60THEN120
130 K=PEEK(59471)
140 A(Q)=K
150 NEXTQ
```

A large number of data inputs from external devices fall into this catagory of sampling at regular time intervals. Intervals in Basic being as small as 1/30 second and in machine code 1/25000 of a second. In some cases instead of sampling the input register at regular

intervals one wants to read and store every data input. This requires that data on the input lines is latched into the input registers by a pulse on the CA1 line. Every time data is latched in by a pulse on this line, the computer reads and stores that data. As an example, an ASCII encoded keyboard is connected to the user port, a key could be pressed at any time, but since the timing and input character is unknown it is impossible to use a programmed wait. It is also unlikely that the data will be present on the input lines for very long and the duration could be variable. If the duration is short a scanning program may miss the data, if the duration is long then the same data will be recorded more than once. The methods looked at so far are obviously unsuitable for this purpose. Each data input is accompanied by a pulse on the CA1 line to latch the data into the input register. Every time there is an active transition on this line bit 1 in the interrupt flag register is set, one can test for an input by testing if that flag is set. The interrupt flag register is located at address 59469 decimal and the setting of this flag can be detected by one of the following two lines of program causing the processor to wait for the flag to be set;

```
100 IF PEEK(59469) AND 2 THEN 110 : GOTO100
      or
100 WAIT 59469,2
```

The CA1 flag is set by an active transition on the CA1 line, this can be either a negative or positive transition depending on the contents of bit 0 of the peripheral control register. If set to 0 then a negative transition sets the flag, a negative transition is one where the voltage on the CA1 line falls from +5 volts to ground. A positive transition will set the flag if bit 0 of the PCR is set to 1. Which transition is chosen depends on the external circuitry and can by set be one of the following two program lines:

```
100 POKE 59468,PEEK(59468) AND 254
```
sets bit 0 of PCR to 0 for negative transition.

```
100 POKE 59468,PEEK(59468) OR 1
```
sets bit 0 of PCR to 1 for positive transition.

When the correct transition occurs on the CA1 line, bit 1 of the interrupt flag register is set, and will remain set until Data register A with handshake control is read or written to. This register located at address 59457 is used instead of the input register at 59471 whenever inputs are latched in under control of line CA1. Whether a transition on CA1 causes data on the input lines to be latched or not depends on whether bit 0 of the Auxiliary control register is set. If the contents of bit 0 of the

ACR is a zero then a transition on CA1 will not cause data on the input lines to be latched into the input register. If the contents of bit 0 of the ACR is 1 then data will be latched thus;

```
100 POKE 59467,PEEK(59467) AND 254
```
inputs not latched

```
100 POKE 59467,PEEK(59467) OR 1
```
inputs latched by a CA1 transition

When using CA1 as a handshaking line, bit zero of both the peripheral control register and the auxiliary control register must be set to the right level before any inputs take place. The following program is an example of how data could be input from an external keyboard to form a string A$.

```
10 POKE 59467,PEEK(59467)OR1:REM LATCH INPUT
20 POKE 59468,PEEK(59468)AND254
25 REM NEGATIVE TRANSITION ON CA1
30 POKE 59459,0:REM SET PA0-7 AS INPUTS
100 WAIT 59469,2:REM WAIT FOR SETTING OF CA1 FLAG
110 K=PEEK(59457):REM READ INPUT, RESET CA1 FLAG
120 K$=CHR$(K)
130 A$=A$+K$:REM ADD INPUT TO STRING A$
140 IFK=13 THEN 200:REM END IF CARRIAGE RETURN
150 GOTO100
200 END
```

Since data is unlikely to come from the keyboard faster than two or three characters per second this Basic program would be adequate. The program could even handle data from a slow speed paper tape reader(the output from this device is identical to that from a keyboard) connected to the user port. If the paper tape reader's speed is gradually increased, data will start to be lost at a point where the input frequency exceeds the minimum execution time of the input program loop. If data is to be input to the PET at high frequency - greater than about 10 bytes per second - then the input program must be written in machine code. Using a machine code subroutine to perform the data input function in a Basic program poses several problems. Unless data is processed by the subroutine, the input must be in descrete blocks of, say 255 bytes, with a delay between each block sufficient to allow the Basic program to process the last block of data. Each data block must be stored in an area unused by Basic from which it can be accessed by the Basic program with a series of PEEK commands. Another requirement is that the computer must not be interrupted during data transfer otherwise data will be lost. This is very important on the PET since the

machine is interrupted sixty times a second as part of the keyboard scanning routine. The interrupt can be disabled by having the first instruction of the machine code subroutine an interrupt disable instruction. Similarly the last instruction must restore the machine's capability of being interrupted. The following is a machine code version of the previous Basic program. It is designed to be located in the area used by cassette #2 input buffer, data input by the program is stored in the top 256 bytes of RAM.

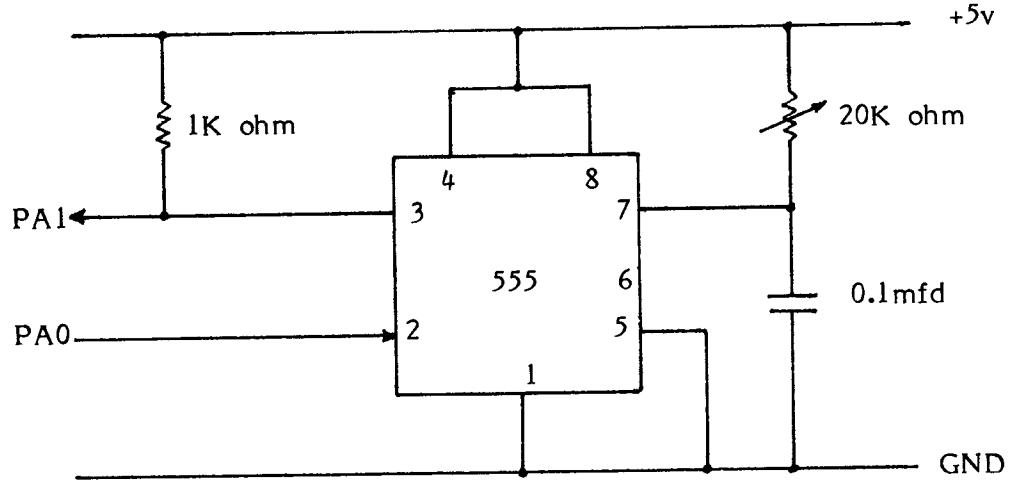
```
033A   78           SETUP    SEI
       AD 4B E8              LDA E84B
       09 01                 ORA 01
       8D 4B E8              STA E84B
       AD 4C E8              LDA E84C
       29 FE                 AND FE
       8D 4C E8              STA E84C
       A9 00                 LDA 0
       8D 43 E8              STA E843
       A2 00                 LDX 00
       AD 4D E8    TESTCA1   LDA E84D
       29 02                 AND 02
       F0 FA                 BEQ TESTCA1
       AD 41 E8    READ      LDA E841
       9D 00 1F              STA 1F00,X
       E8                    INX
       D0 F1                 CLI
       60                    RTS
```

This subroutine can be called by the main program with a SYS 826 command (assuming that the subroutine is located at decimal 826 upwards). Care must be taken that the area in which data is stored is not also required by Basic, this can be prevented by resetting the highest RAM address pointer. Thus to set aside the top 256 bytes of memory the following two commands must be executed at the beginning of the Basic program: POKE 52,255 and POKE 53,126. The Basic program can then access this data and store it as a 255 element array with the following line:

```
100 FOR X=1 TO 255:A(X)=PEEK(7936+X):NEXT X
```

One point to watch when using a subroutine which disables the scan interrupt is that it also stops the jiffy clock. This could cause problems if you are using the clock for any time control purpose. The only cure is to determine the time taken to run the machine code subroutine and add this to the contents of the jiffy clock register in locations 153 and 154 (in old ROMs 517

and 518). Machine code subroutines for data inputs are also useful when precision timing is required, accuracies in the order of ten microseconds can be achieved. This is the kind of precision timing required in the measurement of pulse widths or transient event. A useful application requiring this kind of input is measuring the position of a potentionmeter wiper arm, the potentiometer being part of a position sensing feedback or a joystick input device. Although this may seem like an analog to digital conversion problem there is a far easier solution involving the use of a 555 timer IC. A pulse input to the 555 is output after a delay, the length of which is proportional to the values of an R/C network. By varying the resistance value one can vary the delay time.



The output or trigger pulse comes from PAO on the user port, the input pulse goes to PA1. The following program measures the delay time which is proportional to the current position of the potentiometer arm.

```
03E0    COUNT                       :delay time
033A    78          START           SEI
        A9 01                       LDA 01
        8D 43 E8                    STA E843
        8D 4F E8                    STA E84F
        A9 00                       LAD 00
        8D E0 03                    STA COUNT
        8D 4F E8                    STA E84F
        A9 01                       LDA 01
        8D 4F E8                    STA E84F
        EE E0 03    TEST            INC COUNT
        AD 4F E8                    LDA E84F
        29 02                       AND 02
        F0 07                       BEQ TEST
        58                          CLI
        60                          RTS
```

The program can be run with a SYS(826) and the delay value obtained with a PEEK 992, note however that the machine will crash if no input is obtained on PA1.

It is frequently undesirable to halt the processor while waiting for an input especially in real time control applications. This can be overcome by using the system interrupts. The best method is to add an extra subroutine into the keyboard scanning interrupt routine, the input port will then be automatically scanned sixty times a second. This is especially useful in applications involving the counting of slow but unpredictable events such as those occurring in many biology and psychology experiments. For example the computer is being used to control the environment of an animal cage and we want to measure the activity of the animal. This is done by counting the number of times it breaks a light beam crossing the cage. The animal may spend long periods of time asleep and thus not cause any interruptions of the light beam. It is not therefore practical to have the processor wait for an input, since while waiting it is unable to perform its normal function of controlling the cage environment. The problem is overcome by scanning the current state of the photodetector as part of the keyboard scanning routine initiated sixty times a second by the scan interrupt. In this example the photodiode is connected to line PA0 on the user port via a Schmitt trigger circuit acting as a level detector, so that line PA0 goes to a high state only when the light beam is interrupted. The following program counts the number of times the beam is interrupted:

```
03F0    COUNT                   :total number of beam

03F1    COUNT+1                 :interrupts.

03F8    LAST                    :last input state

033A    A9 00       START       LDA 0
        8D 43 E8                STA E843
        AD 4F E8                LDA E84F
        29 01                   AND 01
        F0 16                   BEQ EXIT
        CD F8 03                CMP LAST
        F0 16                   BEQ END
        8D F8 03                STA LAST
        18                      CLC
        6D F0 03                ADC COUNT
        8D F0 03                STA COUNT
        A9 00                   LDA 0
        6D F1 03                ADC COUNT+1
        8D F1 03                STA COUNT+1
```

```
A9 00       EXIT        LDA 0
8D F8 03                STA LAST
4C 2E E6    END         JMP E62E
```

When loaded into memory this program is started by putting the beginning address into the IRQ RAM vector. If the subroutine is located at hex 033A and upwards the following two Basic commands would be used to start the routine:

in new ROMs - POKE 144,58:POKE 145,3

in old ROMs - POKE 537,58:POKE 538,3

The subroutine will now be automatically executed every sixtieth of a second without being called from Basic program. The results are accessible at any time by PEEKing the contents of COUNT and COUNT+1. When using a program which is part of the interrupt scan routine care must be taken to avoid using the interrupt disable command in another subroutine or disabling the scan interrupt input PIA 1, both these will stop the program.

In some applications it is desirable to use the CA1 input to generate an interrupt rather than use any of the methods looked at so far. Using an external interrupt onto the IRQ line is one of the most difficult ways of inputting data into the PET and should in my experience be used only when absolutely necessary. The reason for this caution is that it is very easy to crash the system with an external interrupt. Also to use the IRQ line it is best if all normal system interrupts are disabled, this means that the keyboard, system clock (TI), tape decks and IEEE port will not function.

Normally the CA1 line does not act as an interrupt but just latches data from the input lines into the input register. For CA1 to function as an interrupt the correct flag in the interrupt enable register must be set. This flag is bit 1 of location 59470 and can be set with a POKE 59470,131, note- bits in this register can only be set if bit 7 is also set. Since the interrupt is generated by the setting of the CA1 flag, the active transition of this line must also be selected by writing a 0 or 1 into bit 0 of the peripheral control register. Having performed these two operations any input on the CA1 line will generate a system interrupt. The PET will stop and jump to an interrupt servicing routine whose address is pointed to in locations 144 and 145 (in old ROMs 537 and 538),however without a user generated routine the PET will crash. The interrupt routine can be located in any area of protected memory, eg. the second cassette buffer. The only requirement is that the last instruction is a jump to the system interrupt subroutine

at hex E61B (in old ROMs E67E). Every time there is an interrupt the user written interrupt handling routine will be performed, this is the source of most problems encountered in using an external interrupt, the reason being that interrupts are generated by more than one device within the PET. The operating system thus has to be able to determine which device generated the interrupt, and the user port is not a recognised system interrupt. A user port interrupt will often cause the machine to crash, also user interrupt handling routines must be able to determine the source of the interrupt. If this is not done then the 60Hz keyboard scan interrupt will have the same effect as a user port interrupt. One way round this problem is to connect the interrupt line to one of the input lines on the port and on each interrupt test if that line has changed state. Alternatively other sources of interrupts can be disabled, the keyboard scan interrupt is disabled with a POKE 59411,58 stopping the keyboard being used and halting the real time clock. The scan interrupt can be restored to its normal function only by executing the following command within a program:

```
100 POKE.59411,61
```

On dynamic RAM machines there is no need to use the IRQ line since the NMI interrupt is available. The NMI interrupt has a higher priority than the IRQ, meaning that an interrupt on the NMI line is executed in preference to one on the IRQ even though they may occur simultaneously. The NMI line can be accessed on the memory expansion connector, a processor interrupt will result from a positive going pulse on this line. The processor will jump to an interrupt subroutine whose address is stored in the NMI RAM vector, locations 148 and 149. Unless the subroutine disables the interrupts they will occur normally and will only affect the execution of the NMI interrupt subroutine by causing a delay every sixtyth of a second. Use of the NMI interrupt is highly recommended in any application involving asynchronous inputs via the user port.

Handshaking on the 6522.

Handshaking is a term used to describe methods of ensuring the synchronisation of input and output pulses between the computer and an external device. There are two handshaking lines on the user port. The CA1 line functions as an input only, acting either as an interrupt to the PET system or latching data currently on the input lines into the input register. The CA1 line is a suitable handshaking line when data is coming from an external source into the PET. When the PET is the originator of data then it must also have an output

handshaking line, for this purpose one can use the CB2 line. The CB2 line can function in either an input or output mode, the mode being determined by the contents of bit 7 of the peripheral control register. If bit 7 is a zero then CB2 acts as an input, if a one then as an output. There are four different input modes and four different output modes, these are determined by the contents of bits 5 and 6 of the PCR. The CB2 line can also act in a free running or serial output mode under control of the 6522 internal shift register.

Only the manual output modes and the free running / serial modes are of practical use on the PET. The remaining two modes are concerned with the setting of the CB2 line by writing to, or reading the B output register. They are of little use since we want to handshake outputs on port A, the user port. The simplest method of outputting on the CB2 line is to toggle it off and on under manual or program control. Before doing this the shift register must be disabled by setting bits 2, 3 and 4 of the Auxiliary control register to zero. This can be done from Basic with :

```
POKE 59467, PEEK (59467) AND 227
```

If bit 5 of the PCR is set to zero then the CB2 line is low and if set to one then CB2 is high, bits 6 and 7 of the PCR are set to one in both modes. To set CB2 high from Basic the following command can be used:

```
POKE 59468, PEEK(59468) AND 31 OR 224
```

CB2 can be set low with

```
POKE 59468, PEEK(59468) AND 31 OR 192
```

To handshake a byte of data from the eight user port lines to an external device the data must be loaded into the output register. Then the CB2 line must change state, from say low to high, signalling to the external device that data is present. If the other device is another PET then CB2 could be connected to the CA1 line of the second PET. A transition on the CB2 line would latch the data on the parallel lines into the second PETs input register. This principle applies to any external device using a 6522 or PIA type chip. The following Basic program will do this and since the output is parallel fairly high data transmission rates can be achieved even with a basic program.

```
100 POKE59459,255 :REM SET DDR 0-7 AS OUTPUTS
110 POKE59467,PEEK(59467)AND227 :REM DISABLE SHIFT REG
120 POKE59468,PEEK(59468)AND31OR192 :REM SET CB2 LOW
130 POKE 59471,X :REM WRITE VARIABLE X TO ORA
140 POKE 59468,PEEK(59468)AND31OR224 :REM SET CB2 HIGH
```

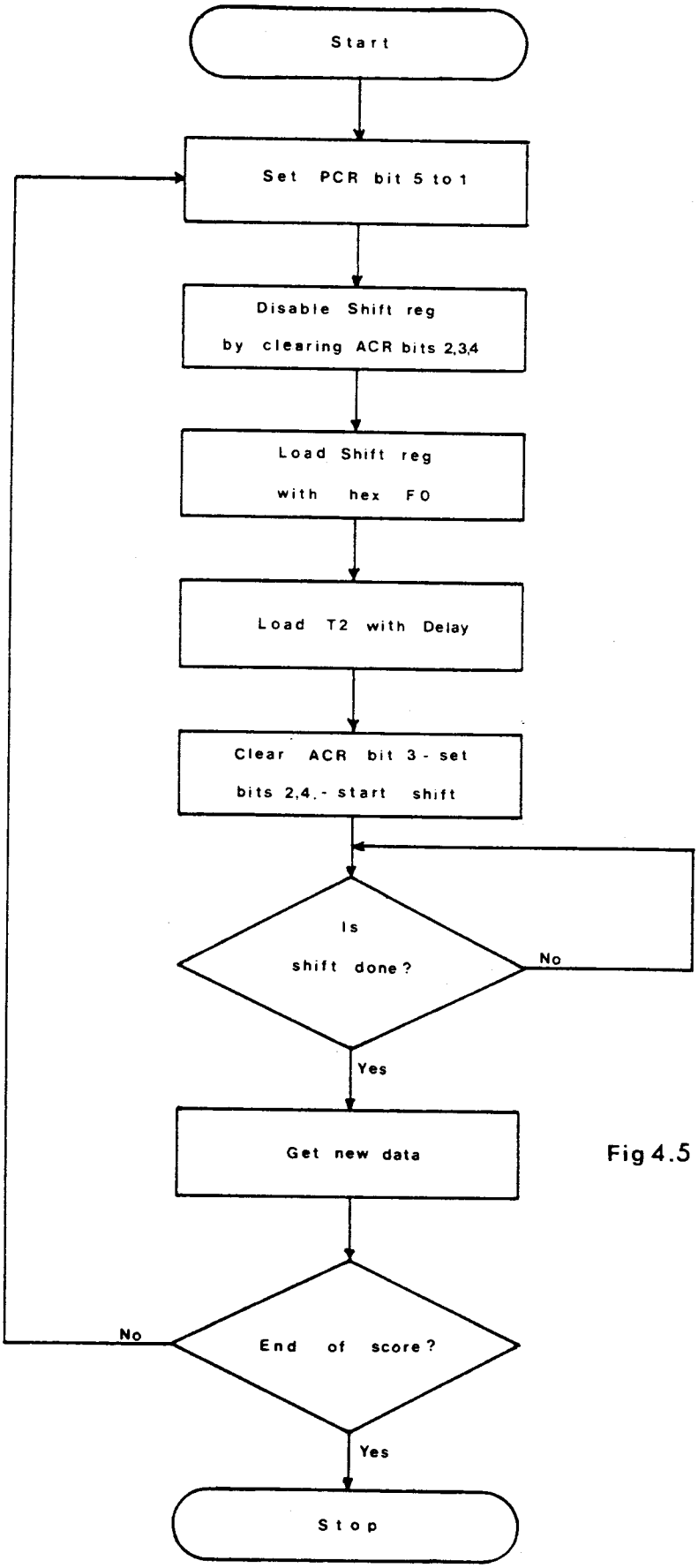


Fig 4.5 Flow diagram of Music program

The 6522 has an internal parallel input serial output shift register. Data is loaded into the shift register in the same way that it would be loaded into any other eight bit register. The data is then shifted out onto the CB2 line under control of either timer 2, the system clock or an external clock. Of the four serial output modes the free running mode is the simplest, and the only mode easily controlled from Basic. In this mode the shift register acts in a cyclical manner with the output from bit eight being fed back to bit zero. The rate at which data is shifted out onto the CB2 line is determined by the contents of timer 2. This timer is a presettable counter, counting the number of clock pulses. On each clock pulse the counter is decremented, if the contents is zero a pulse is output to the shift register thereby shifting the contents one bit to the right. At the same time the timer is reset to its initial value and the process repeated. In this way a repeated pattern of eight bits can be shifted out onto the CB2 line at a particular frequency and totally independent of processor control. The output will continue until either the timer or shift register are changed, or disabled. By loading the shift register with 00001111 and the timer with 255 a square wave can be output on CB2 with a frequency of 490Hz. The highest frequency is obtained by setting the shift register to 01010101 and the timer to 1 giving a square wave output of 500KHz. This free running output on the CB2 line is a useful source of clock pulses for an external device, ensuring full synchronisation with the PET timing. On the more entertaining level this mode can be used to create a simple music generator, by varying the output frequency on CB2. The following is a machine code program to do this:

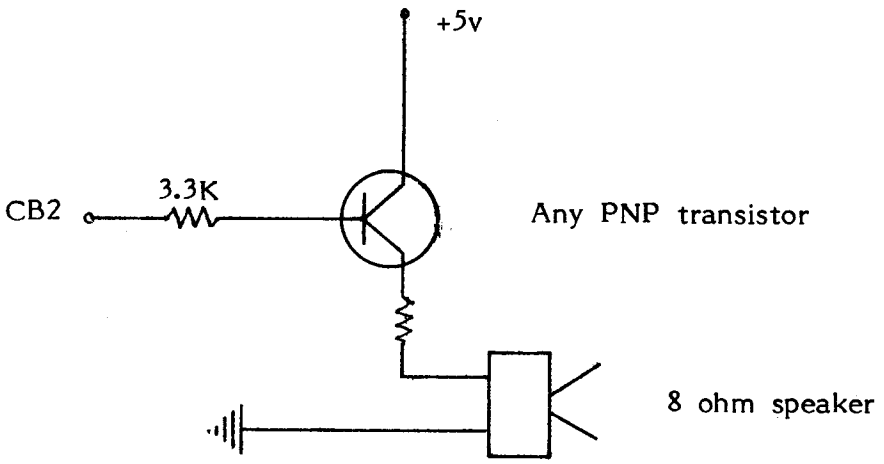
```
                        ;SYSTEM LOCATIONS
                         ACR = $E84B
                         SR = $E84A
                         TIM2 = $E848
                        ;VARIABLES
1900                     YTEMP          :temporary Y register
1901                     TEMPO          :delay count for tempo

1910    A9 10    SETUP   LDA #10
        8D 4B E8         STA ACR
        A9 F0            LDA #F0
        8D 4A E8         STA sr
        A0 00            LDY #0
191C    B9 00 E8 GETNOTE LDA NOTE,Y
        8D 48 E8         STA TIM2
        F0 20            BEQ END
        C8               INY
1925    B9 00 1A GETDUR  LDA DUR,Y
        C8               INY
1929    AA       DUR     TAX
        8C 00 19         STY YTEMP
```

```
192D    A9 03       LOOP      LDA #03    :adjust for tempo
        8D 01 19              STA TEMPO
1932    A0 FB       LOOP1     LDY #FB
1934    88          LOOP2     DEY
        D0 FD                 BNE LOOP2
        CE 01 19              DEC TEMPO
        D0 F6                 BNE LOOP1
        CA                    DEX
        D0 EE                 BNE LOOP
193F    AC 00 19    RESTORE   LDY YTEMP
        D0 D8                 BNE GETNOTE
1944    A9 00       END       LDA #0
        8D 4B E8              STA ACR
        8D 4A E8              STA SR
        8D 48 E8              STA TIM2
        60                    RTS

1A00                ;START OF SCORE TABLE
                     NOTE1,DUR1,NOTE2,DUR2,NOTE3..ETC
```

The circuit used to generate the sound is very simple consisting of a single transistor amplifier and a small 8 ohm speaker.



The shift register has been designed to allow the CB2 line to act as a synchronous serial communications port. This is the function of the remaining three shift register output modes.

The first mode is similar to the free running mode, data being shifted out under control of timer 2. Instead of recirculating indefinately only eight shift pulses are generated, then the shift register is automatically disabled. At the same time that the shift register is disabled the shift register interrupt flag is set. The CB2 line then goes to a state determined by the contents

of bit 5 of the PCR. In any practical application this output mode must be controlled from a machine code program. The flow diagram for such a program is shown in Figure 5. This outputs data in a serial format, if the timing and formatting is correct this could be used by an external device such as a terminal. An interesting feature of this mode is that the shift pulses generated by timer 2 are output on line CB1, the cassette read line. It can be accessed by the user from the top connections of the user port or from the second cassette port (note that this is the reason why the cassette will not function in any of the CB2 output modes). This is a useful feature since it allows serial data output on the CB2 line to be synchronised with the system clock thereby opening up a whole range of possible low cost I/O configurations. The remaining two shift register output modes on the CB2 line are very similar except that the shift timing is derived from different sources. One comes from the 1MHz system clock, the other from an external clock connected to the CB1 line. All four shift register output modes are controlled by the contents of bits 2, 3 and 4 of the Auxiliary control register-ACR-which can be loaded by ANDing the contents with decimal 227 and then ORing it with the required ACR value. In the free running mode, which is the only mode that can be realistically controlled from Basic, the ACR can be set with the command:

```
POKE 59467,PEEK(59467) AND 227 OR 16
```

If the other ACR functions are not used POKE 59467,16 will suffice - the following is a summary of the four modes:

ACR bits 4 3 2	Mode	OR value decimal
1 0 0	Free running under control of T2	16
1 0 1	Shift out 8 bits; shift rate controlled by T2 shift pulses generated on CB1	20
1 1 0	Shift out at system clock rate	24
1 1 1	Shift out under control of an external clock input on CB1	28

The CB2 line can also act as an input, there are four input modes under control of PCR bits 5, 6 and 7. With the shift register disabled it is however only practical to use two of these modes on the PET. One mode detects a negative transition on the CB2 line, the other a positive transition. An input sets bit 3 of the Interrupt flag register. An input on the CB2 line could be used as a system interrupt by setting bit 3 of the Interrupt enable register. This will however encounter the same problems as an interrupt on the CA1 line and is thus probably best avoided. As in the output mode the

shift register can be disabled by setting bits 2, 3 and 4 of the Auxiliary control register to zero. This can be done with the Basic command:

```
POKE 59467, PEEK(59467) AND 227
```

To detect an input with a negative transition one must first set bits 5, 6 and 7 of the PCR to zero with the command:

```
POKE 59468, PEEK(59468) AND 31
```

To detect a positive transition bit 6 of the PCR is set to one and bits 5 and 7 set to zero with the command:

```
POKE 59468, PEEK(59468) AND 31 OR 64
```

The result of either of these two transitions can be detected by testing if bit 3 of the Interrupt flag register is set with one of the following commands:

```
100 IF PEEK (59469) AND 8 THEN 110
110 .....
or 100 WAIT 59469,8
```

Having detected a transition the interrupt flag must be reset before another transition can be detected. The reset is done by reading the port B I/0 register (note care should be taken not to write to this register) this can be done with the command:

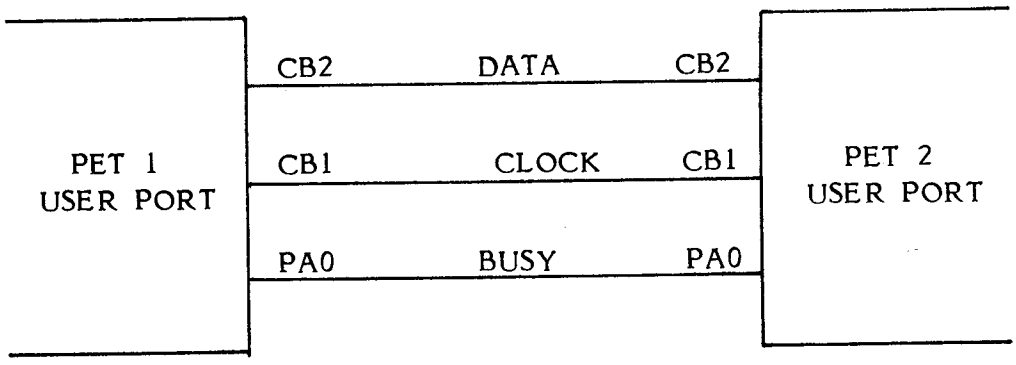
```
Q = PEEK (59456)
```

The CB2 line can also be used as a serial input using the shift register to convert the stream of pulses into eight bit blocks of data. There are three modes of serial input each using a different source of shift pulses. As with serial output these sources are: timer 2, the system clock, and an external clock input on CB1. Except for the last mode the shift pulses are output on the CB1 line. In the timer 2 mode the shifting rate is controlled by the contents of T2. The time between transitions on the output clock on CB1 is a function of the contents of T2 and the 1MHz system clock. In the system clock input mode data is shifted onto the shift register at half the system clock rate or 500KHz. The shifting operation in both modes is initiated by either reading or writing the shift register in location 59466. The data is shifted into the shift register on the trailing edge of each shift pulse. The first bit of the input data being shifted into bit zero is the most significant bit. Also data transitions should occur before the leading edge of the shift pulse. After eight shift pulses the shift register interrupt flag, bit 2 of

the interrupt flag register, will be set and the output clock pulses on CB1 will stop. To shift data in under control of an external clock CB1 becomes an input and data is shifted in during the first system clock cycle following the leading edge of the CB1 shift pulse. As with the other serial modes data is shifted into bit 0 of the shift register first. Unlike the other modes the shift register is not disabled, though the interrupt flag is set, after 8 shift pulses. The interrupt flag can be reset by reading the shift register. When using an external clock data transfer rates should thus be kept fairly low. All shift register input modes are best controlled by a machine code program unless data rates are very slow. These modes are controlled by bits 2, 3 and 4 of the Auxiliary control register and can be summarised as follows:

ACR bits 4 3 2	Mode
0 0 1	Shift in under control of timer 2, shift pulses output on CB1.
0 1 0	Shift in at system clock rate, shift pulses output on CB1.
0 1 1	Shift in under control of external input on CB1.

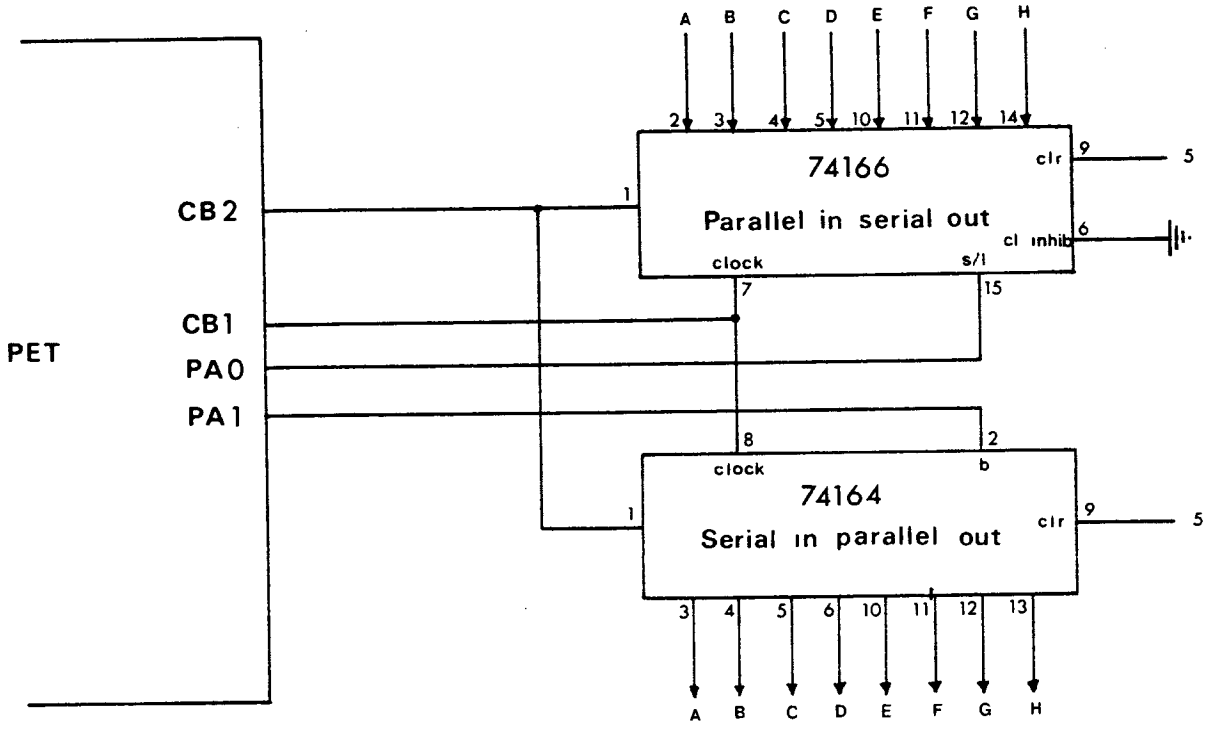
The serial I/0 capability of the CB2 and CB1 lines can be used as the basis for a range of interesting and useful I/0 configurations both between PETs and between PET and peripheral devices. One application is to use these lines for data and program communication between two machines. To do this the corresponding CB1 and CB2 lines are connected and also one of the user port lines, say, PA0 on each machine. The CB2 line is used as a bidirectional data communications line while the CB1 line is the clock line used to synchronise data transfer betwen the two computers. The line between the two PA0s is the "busy" line and is used to signal to the transmitting machine that the receiving machine is ready for data input.



The software required to control such a communications system is not complex and could if one were prepared to accept a very slow and inefficient system be written in Basic. This software relies on two rules, one for the transmitter and one for the receiver. The rule for the transmitter is that data output is under control of timer 2 and does not begin transmitting data until the busy line goes "low". The rule for the receiver is that data is shifted into the receiving machine under the control of an extenal clock. This is derived from the CB1 shift pulse output on the transmitting machine, thereby ensuring that the data is fully synchronised. The "busy" line should be kept "high" until th receiving machine is ready to accept an input. This fairly simple method of communicating between two machines could probably be expanded to allow the construction of small networks of PETs by using a separate "busy" line for each computer. The software for either two machines communications or network communications is best written in machine code and could be called as a subroutine from a Basic progam when required. Or it could be incorporated into the scan interrupt routine for automatic operation. More ambitiously instructions could be added to Basic by calling the machine code subroutine from a section of code added to the CHARGET subroutine in page zero of memory.

The serial I/0 capability of the CB2 line and its accompanying clock pulses on CB1 can be used to greatly expand the number of I/0 lines available on the user port with only the minimum of extra circuitry. This technique is especially useful in applications requiring a great many single line inputs and outputs. For example input switches and status lamps, where data inputs or outputs are unlikely to change very frequently. The method relies on inputting or outputting all data in a serial form via the CB2 line. This data is then converted to or from parallel form by an eight bit shift register, data being shifted in or out under control of clock pulses from CB1. The shift enable input is derived from one of the user port lines, allowing up to eight blocks of shift registers where each block has either an input or an output function. With one shift register per block this gives a maximum of 64 I/0 lines. A suitable integrated circuit for outputs would be a 74164 and for inputs a 74166. The number of input or output lines can be increased by chaining two or more shift registers together under control of a single enable line. The serial input of one register being connected to the serial output or last parallel output line of the next register.

With this technique the software required to input and output data is very simple and easily written in Basic. The following is a Basic program to output a variable X and the circuit used by the program:



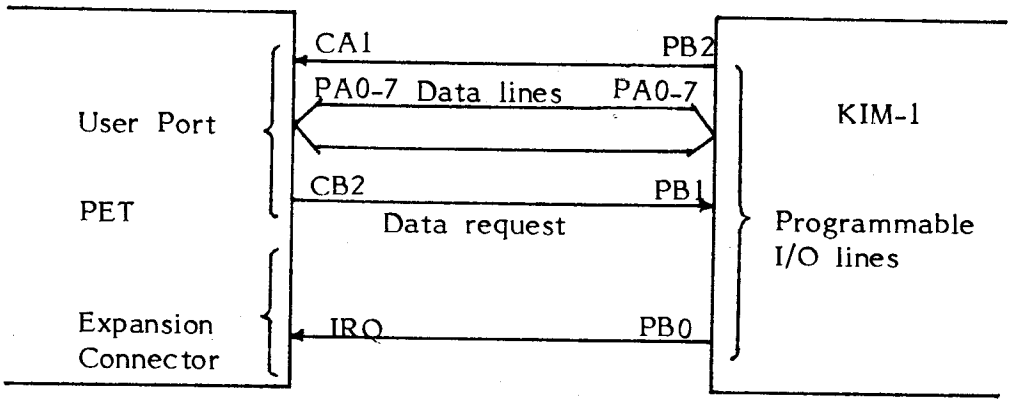

```
10 REM program to output variable X
100 POKE 59459,2  : REM set DDR for PA1 as an output
110 POKE 59471,2  : REM output chip enable on PA1
120 POKE 59464,64 : REM set timer 2, value optional
130 POKE 59467,PEEK(59467)AND 227 OR 20 : REM set ACR for
                                          SR output under T2
140 POKE 59466,X  : REM write variable into shift register


10 REM program to input variable X
100 POKE 59459,1  : REM set DDR for PA0 as an output
110 POKE 59471,1  : REM output chip enable on PA0
120 POKE 59467,PEEK(59467)AND227 OR 4  : REM set ACR for
                                         input under T2
130 WAIT 59469,4  : REM wait for SR interrupt flag set
140 X=PEEK(59466) : REM read contents of SR
```

Obviously one need not use these particular user port lines, any line will do, in the two examples program lines 100 and 110 must be altered accordingly. Line 100 sets the data direction register for all the input and output enable lines and need only be done once at the beginning of the program.

PET - KIM Data Handshaking Via The User Port.

The following application is an example of how the user port can be used to interface the PET to another computer, in this case a Kim 1. The application involves transferring blocks of 128 bytes of data from the Kim to the PET once every ten seconds, with the transfer lasting about 100 milliseconds. The eight lines of the User port are connected, together with the two handshaking lines CA1 and CB2, to ten of the Kim I/0 port lines. A further I/0 port line on the Kim is used to generate an interrupt request signal and is connected to the IRQ line on the PET memory expansion connector.



The interrupt is used to ensure that the PET services the Kim request to transfer data as rapidly as possible, thereby ensuring the minimum amount of time spent by either processor waiting for the other. The flow diagram of the handshaking routines of both processors is shown in Figure 4.6. To ensure that the routine is executed as a result of an interrupt generated by the Kim, and not by the PET system interrupts, the internal vector pointers must first be reset. This is done by a small subroutine - INTDIS - which is called at the beginning of any Basic control program by a SYS(839), the vectors can be reset by another subroutine - INTEN - called by SYS(826).

The subroutine INTDIS also performs the function of resetting the top of memory pointers to leave a 128 byte block of unused memory space at the top of memory for data storage. The program is fairly short and can be

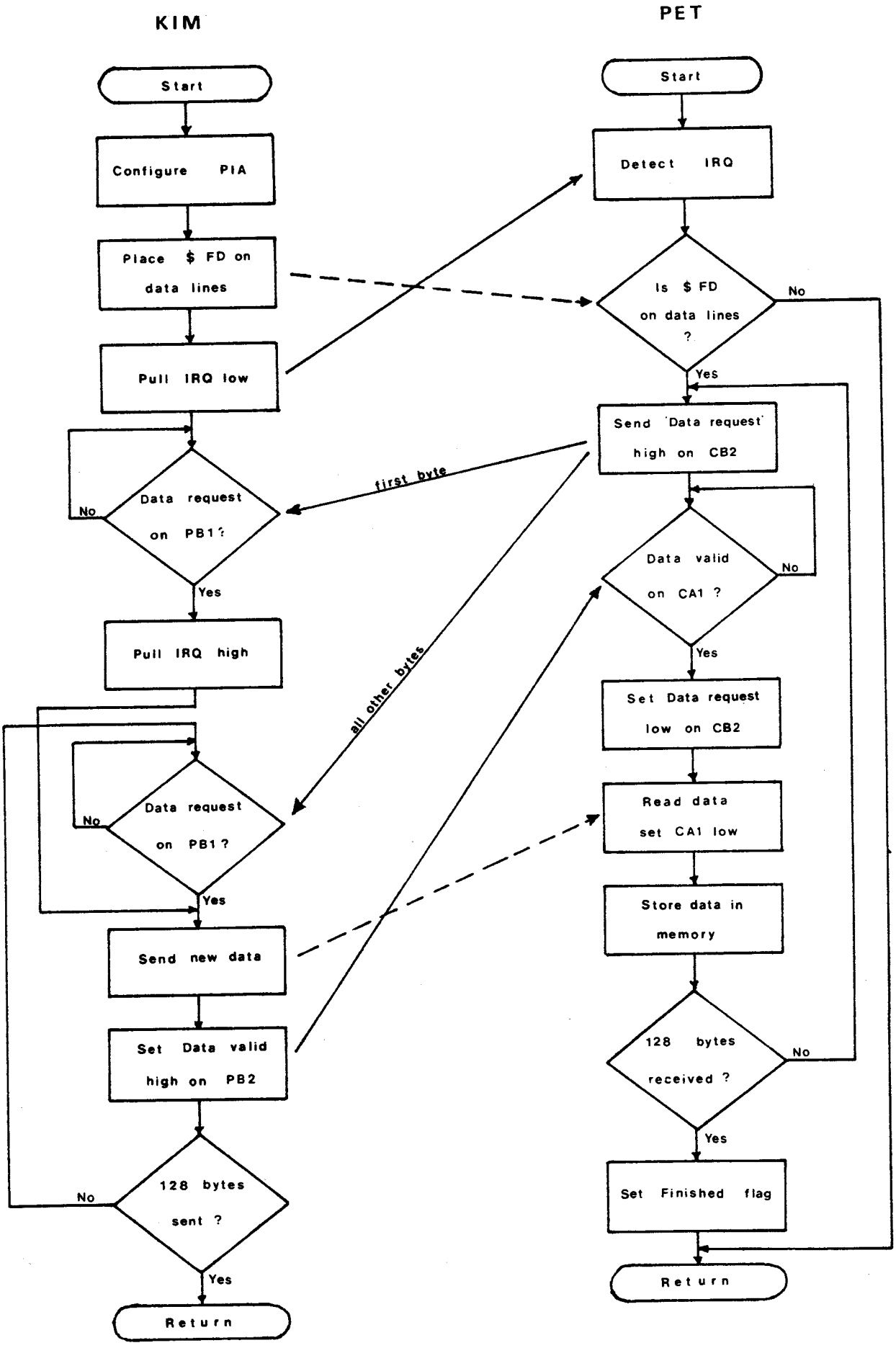


Fig 4.6 Kim - PET Data Handshaking

```
033A  78          INTEN   SEI           :re-enable system interrupt
      A9 85               LDA $85       :low order byte of vector
      8D 19 02            STA $0219
      A9 E6               LDA $E6       :high order byte of vector
      8D 1A 02            STA $021A
      60                  RTS
0347  78                  SEI           :disable system interrupt
      A9 5C               LDA $5C       :low order byte of vector
      8D 19 02            STA $0219
      A9 03               LDA $03       :high order byte of vector
      8D 1A 02            STA $021A
      A9 7F               LDA $7F       :low order top of memory
      85 86               STA,z $86     :pointer location 134
      A9 1F               LDA $1F       :high order top of memory
      85 87               STA Z $87     :pointer location 135
      58                  CLI
      60                  RTS
035C  AD 41 E8  DATAIN    LDA $E841     :data handshake routine
      C9 FD               CMP $FD       :start read data lines
      DO 29               BNE END       :if not $ FD goto END
      AO OO               LDY $O        :set index to zero
      A9 E1       DATA1   LDA $E1       :if FD on data lines set
      OD 4C E8            ORA $E84C     :CB2-data request-high
      8D 4C E8            STA $E84C
      A9 02       DATA2   LDA $02       :wait for data valid on CA1
      2D 4D E8            AND $E84D
      FO F9               BEQ DATA1     :if not goto DATA1
      A9 DF               LDA $DF
      2D 4C E8            AND $E84C     :pull CB2 low and remove
      8D 4C E8            STA $E84C     :data request
      AD 41 E8            LDA $E841     :input data and store in top
      99 80 1F            STA,Y $1F80:of memory using index pointer
      C8                  INY           :increment index
      C0 80               CPY $80       :is index = 128 if not then
      DO DE               BNE DATA1     :goto DATA1
      A9 FF               LDA $FF       :set flag to 255 and store
      8D 8F 03            STA $038F     :in location 911
      4C 85 E6    END     JMP $E685     :jump back to Basic
038F  :           END FLAG POINTER
```

Summary of the Registers in the 6522.

Parallel port PB
59456 Hex E840

7	6	5	4	3	2	1	0
DAV in	NRFD in	RETRACE in	Cass #2 Motor	Cassette Output	ATN out	NRFD out	NDAC oin

This register contains the contents of the input and output lines of port B of the 6522. It can be read but should not be written to with the exception of bit four which turns the motor of cassette 2 off and on. Reading this register causes the CB2 interrupt flag to be reset.

Parallel port PA with handshake control
59457 Hex E841

7	6	5	4	3	2	1	0
User Definable I/O Lines							

This is one of two registers which contain the contents of the input and output lines of port A. The two registers are identical except that this register has control over the handshake lines. When data is input using the CA1 line to latch data into the I/0 register, the fact that data has been input is signalled by the setting of the CA1 interrupt flag. This flag is cleared by reading address 59457.

Data direction register for port B
59458 Hex E842

This register should not be used on the PET.

Data direction register for port A.
59459 Hex E843

This register controls each of the eight lines on port A and determines whether they are acting as inputs or as outputs. A one in any of the eight bits of this register sets the corresponding line into the output mode and a zero puts it into the input mode.

Timer 1.
lower order byte 59460 Hex E844
higher order byte 59461 Hex E845

This sixteen bit register is one of two internal timers on the 6522. However this is of no, (or limited) use on the PET since it generates timed interrupts and/or output on line 7 of port B which is the DAV input line.

Timer 1 latch.
lower order byte 59462 Hex E846
higher order byte 59463 Hex E847

This sixteen bit lach is used to store data which will later be loaded into the counter of timer 1, since this timer is not used on the PET the latch is of little use.

Timer 2.
lower order byte 59464 Hex E848
higher order byte 59465 Hex E849

This is the second of the two internal timers on the 6522 and as with timer 1 the majority of its functions are not usable on the PET. The lower order eight bits of timer 2 can be used to generate shift pulses for the internal shift register thus allowing variable speed serial I/0 on the CB2 line. The timer can be loaded by POKEing a value between 1 and 255 into location 59464.

Shift register
59466 Hex E84A

The internal eight bit shift register is a very useful feature of the 6522 since it allows serial data transfer into and out of the CB2 line. This is controlled by either timer 2, the system clock or an external clock. The mode of operation of the shift register is controlled by the contents of bits 2, 3 and 4 of the Auxiliary control register. In some modes the completion of the shift operation is signalled by the setting of the shift register interrupt flag in bit 2 of the IFR. The shift register may be loaded by POKEing the data into location 59466.

Auxiliary control register
59467 Hex E84B

7 6	5	4 3 2	1	0
Timer 1 control	Timer 2 control	Shift Register mode control	Port B latch enable	Port A latch enable

The function of the auxiliary control register is to

control the mode of operation of the other 6522 registers. However, on the PET it is only practial to control two of these registers, the shift register and the port A latch enable. Bit 0 is the port A latch enable which when set to 1 allows data to be latched into the input register by a pulse on the CA1 line. When set to zero the input register will directly reflect the data on the input lines. A similar function is performed by bit 1 to control the latching of data into port B, but the contents of this bit should not be altered. Bits 2, 3 and 4 control the operation mode of the shift register. There are eight modes of operation and they are best summarised as follows:

ACR bits 4 3 2	Shift register mode	SR interrupt flag
0 0 0	Shift register disabled	.....
0 0 1	Shift in under control of T2	set after 8 shifts
0 1 0	Shift in under system clock	set after 8 shifts
0 1 1	Shift in under control of external clock pulse	set after 8 shifts
1 0 0	Free running output at rate determined by T2	.....
1 0 1	Shift out under control of T2	set after 8 shifts
1 1 0	Shift out under system clock	set after 8 shifts
1 1 1	Shift out under control of external clock pulse	set after 8 shifts

Bits 5, 6 and 7 control the functioning of the two timers neither of which can be used on the PET. The auxiliary control register can be loaded from Basic by using the following command format --- POKE 59467, PEEK(59467) AND ... OR ... where the dots are variables, the value of which depends on the bits being changed. Thus if changing the shift register to free running mode it would be AND 227 OR 16.

Peripheral control register
59468 Hex E84C

7 6 5	4	3 2 1	0
CB2 control	CB1 control	CA2 control (graphics / lower case)	CA1 control

The peripheral control register controls the functioning of the four handshaking lines on the 6522. All four lines can be controlled by the user on the PET. Bit zero selects which active transition of the CA1 line sets the CA1 interrupt flag. A one in this bit sets the flag on a positive transition (low to high) and a zero sets the flag on a negative transition (high to low). Bit 4 of the PCR performs the same function for the CB1 line, this is the read line for cassette 2 but can be used as

an I/O line if this cassette is not used. CA2 is connected to the character generator and controls whether the display is in the graphics or lower case mode, the display mode can be changed by toggling this line. Though the CA2 line can funtion as both an input and an output, on the PET it can only function in the manual output mode. The display can be put in the lower case mode with a POKE 59468,14 and in the graphics mode with a POKE 59468,12. This is the normal method used where the contents of all the other bits in the PCR are zero if however the CA1, CB1 or CB2 controls are set then they must be masked out with an AND 225, thus to put the display in lower case becomes POKE 59468, PEEK(59468) AND 225 OR 14. The CB2 line is totally under user control and can act as either an input or an output. There are eight modes of operation, four of them can be used on the PET, they can be summarised as follows:

PCR bits	CB2 operation mode
7 6 5	
0 0 0	Input mode sets CB2 interrupt flag on negative transition flag, reset by reading port B register.
0 1 0	Input mode, sets CB2 interrupt flag on positive transition flag , reset by reading port B register.
1 1 0	Manual output mode, CB2 is held low.
1 1 1	Manual output mode, CB2 is held high.

Interrupt flag register
59469 Hex E84D

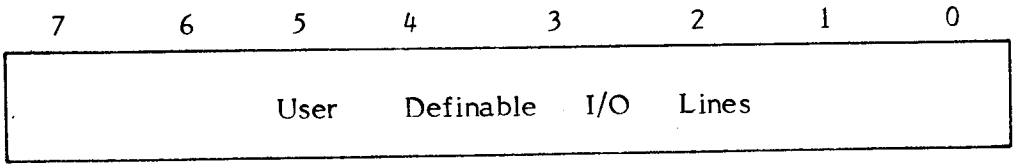
7	6	5	4	3	2	1	0
IRQ status	T1	T2	CB1	CB2	SR	CA1	CA2
interrupt flags							

The four handshaking lines, the shift register and the two timers are all able to generate a system interrupt by setting a bit in interrupt flag register. Providing the corresponding bit in the interrupt enable register is set this will cause an interrupt to be generated. Reading the interrupt flag register will then indicate which register or handshake line initiated the interrupt. The setting of a particular flag will also show when an operation has been completed or a particular event has occurred. As a PET user the most useful flags are the CB2, SR, and CA1, the use of these flags has been dealt with in the review of the relavent registers. Note that bit 7 of this register is not an interrupt flag but shows the current status of the IRQ output to the processor it is only set by a system interrupt it can only be cleared by clearing all the flags in the register.

Interrupt enable register
59470 Hex E84E

When a bit in this register is set and the corresponding bit in the interrupt flag register is also set, then and only then will a system interrupt be generated. In the PET this register should not be used since enabling any of the interrupts will invariably cause a system crash.

Parallel port PA
59471 Hex E84F



This is the second of the two registers containing the contents of the input and output lines of port A. This register has no control over the handshaking lines. The d rection of the data transfer in this port is controlled as in the other port A register by the contents of Data direction register A. Data may be directly read or written into this register using PEEK or POKE commands.

THE IEEE PORT AND 6520s 5

A total of three peripheral I/O chips are used on the PET, the 6522 which we looked at in Chapter 4 and two 6520 PIAs. The primary function of one PIA being to control the keyboad, the other the IEEE 488 port. The 6520 is a simpler version of the 6522, like that chip it has two eight bit bi-directional I/O ports with handshaking lines. It has six internal registers (three for each I/O port) though only four can be directly addressed by the processor at any one time. The internal architecture of this chip is shown in Figure 5.1. The registers are two peripheral registers, two data direction registers and two control registers. Registers are selected by address line 0 and 1, together with bit 2 in the control register thus:

Address lines A1	A0	CRA bit 2	CRB bit 2	Register selected
0	0	1	X	Peripheral register A
0	0	0	X	Data direction register A
0	1	X	X	Control register A
1	0	X	1	Peripheral register B
1	0	X	0	Data direction register B
1	1	X	X	Control register B

Each I/O line on the 6520 can be independently programmed as either input or output by setting the corresponding bit in the data direction register to zero for an input and one for output. The data direction register is first enabled by writing a zero into bit 2 of the control register for the port. Having set the data direction register this bit must be reset to a one before the I/O port can be read or written to. It is not advisable to alter the data direction of the I/O lines on either 6520 in the PET.

The two control registers are the most important registers of the 6520 allowing the processor to control the operation of the four peripheral control lines CA1, CA2, CB1 and CB2, as well as controlling the generation of interrupts and enabling the data direction register. The two control registers, one for each port are

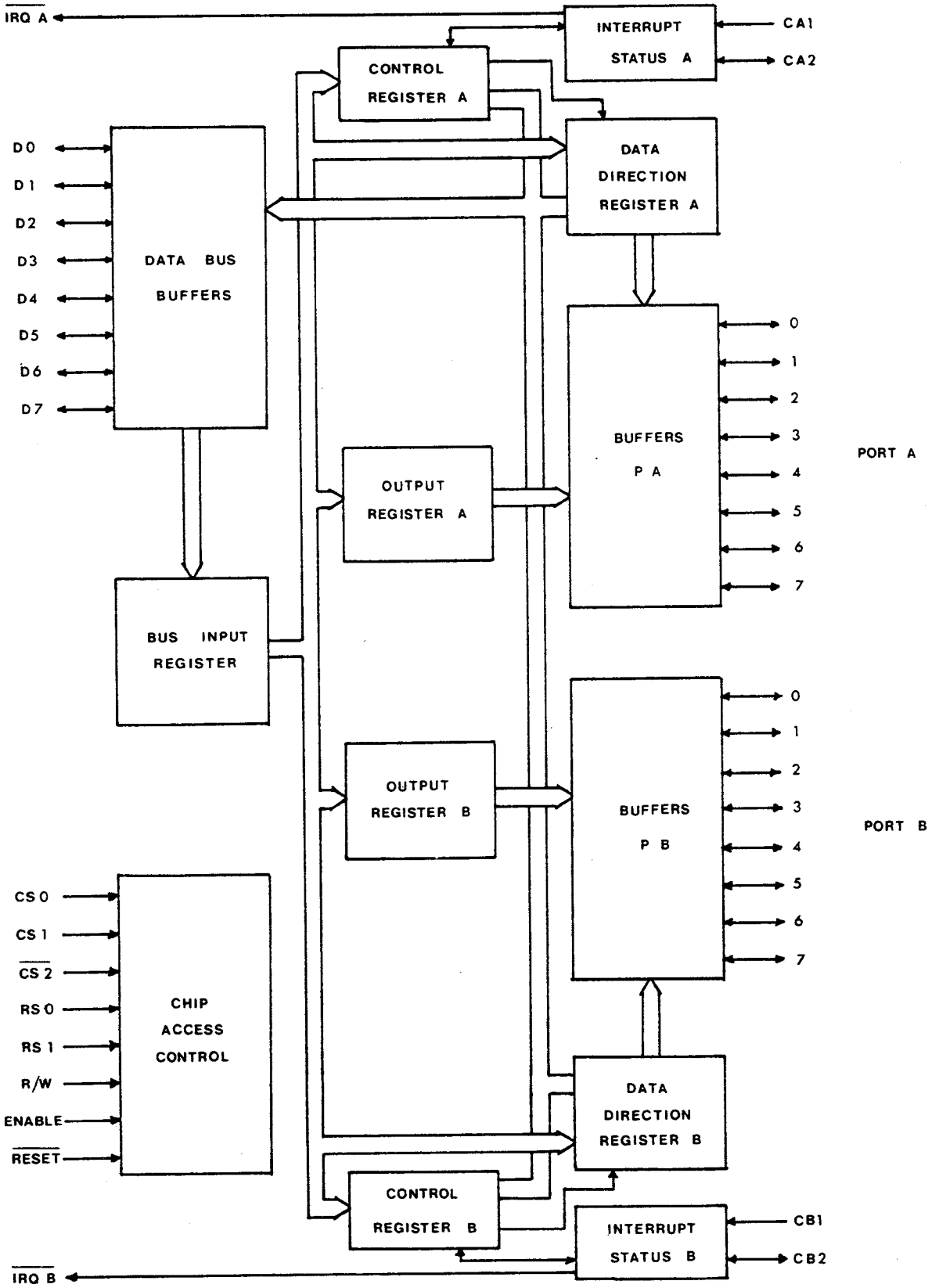


Fig 5.1 6520 Block Diagram

identical and have the following format:

7	6	5 4 3	2	1 0
IRQ 1 A or B	IRQ 2 A or B	CA2 or CB2 control	DDRA or B access	CA1 or CB1 control

Bit 2 is used to select whether the processor addresses the peripheral I/O register or the Data direction register, both registers being located at the same processor address. The interrupt flags in bits 6 and 7 are set by an active transition on the interrupt or peripheral control lines (when programmed as inputs). These flags can not be set by the processor and can be reset only by reading the relavent I/O register.

The CA1 and CB1 lines act as interrupt inputs only, an active transition on one of these lines will set bit 7 of the relevent control register to logic 1. The transition is controlled by bit 1 of the control register, if bit 1 is set to a logic 0 then the interrupt flag is set on a negative transition, if bit 1 is set to 1 then a positive transition will set the flag. The setting of the interrupt flag will cause a system interrupt to be generated on the IRQ line only if bit zero of the control register is set to a logic 1. The IRQ output can be disabled by setting this bit to logic 0. Note that great care should be taken when using system interrupt on the PET, polling techniques being always used in preference.

The CA2 and CB2 lines can act as either totally independent interrupt inputs or as peripheral control outputs, the mode of operation being determined by bit 5 of the port's control register. If bit 5 is set to 0 then CA2 and CB2 are in the input mode, set to 1 they are in the output mode. In the input mode an active transition on one of these lines will set the interrupt flag in bit 6 of the control register. The active transition is selected by bit 4 of the control register, a zero will set the flag on a negative transition, a one will set it on a positive transition. An input on either CA2 or CB2 will result in a system interrupt being generated on the IRQ line unless the interrupts are disabled by setting bit 3 of the control register to zero. If either interrupt flags are set when the relevent bit of control register (either bit 0 or 3) has disabled the interrupt, then enabling the interrupt will immediately cause the IRQ lines to go low and generate an interrupt.

In the output mode CA2 and CB2 are slightly different in their function and must therefore be looked at separately. The CA2 line operates in the output mode when bit 5 of the control register is set to 1. There are three output modes for this line, they are

Fig 5.2 SYSTEM I/O MEMORY MAP

PIA 1 (6520)

E810	Diagnostic Sense	IEEE EOI in	Cassette Sense #2 #1	KEYBOARD ROW SELECT	PA	59408
E811	Tape #1 Input flag	....	Screen blank output (old 8K only) IEEE EOI out CA2	DDRA Access	Cassette #1 Read control CA1	59409
E812	KEYBOARD ROW INPUT					59410
E813	Retrace I flag	....	Cassette #1 motor output CB2	DDRB Access	Retrace interrupt Control CB1	59411

PIA 2 (6520)

E820	IEEE INPUT					59424
E821	ATN I flag	....	IEEE NDAC out CA2	DDRA Access	IEEE ATN in Control CA1	59425
E822	IEEE OUTPUT					59426
E823	SRQ I flag	....	IEEE DAV out CB2	DDRB Access	IEEE SRQ in Control CB1	59427

determined by the contents of bits 3 and 4 of the control register. By setting bit 4 to 1, CA2 can be manually toggeled by clearing or setting bit 3. Putting a zero into bit 3 will set CA2 low, a one in bit 3 will set CA2 high. The second output mode is a pulse output mode in which the CA2 line goes low for one clock cycle after a read peripheral register A operation. This mode can be initiated by setting bit 4 of the control register to 0 and bit 3 to 1. The pulse mode can be used to indicate to a peripheral device that data has been read or used to clock a shift register or counter thereby allowing sequential data input on the I/0 lines. In the third and last mode the CA2 line is set high by an active transition on the CA1 input setting the CA1 interrupt flag. It can be set low again by the processor reading the peripheral A I/0 register. This handshaking mode allows the CA2 line to signal to the peripheral device that it is ready to accept new data. The handshake on read mode can be initiated by setting both bits 3 and 4 of the control register to zero. The output modes of the CB2 line differ in that the pulse output mode occurs when the processor writes data to the peripheral I/0 register B, similarly in the handshaking mode the CB2 line goes low when the processor writes to peripheral I/0 register B.

The operation of the 6520 is reletivly simple compared to the 6522, it has many useful features such as its control of the handshaking lines. Of the two 6520 PIA chips in the PET, the first controls the keyboard and the majority of the lines to cassette 1 as well as the diagnostic input and the retrace interrupt. The second 6520 is devoted entirely to the IEEE 488 I/0 port. The location of these two PIA chips and the function of each bit is shown in Figure 5.2.

The Keyboard.

The keyboard on the PET has 73 keys, 64 print character keys plus 9 function keys(like cursor control and reverse). The keyboard is scanned 60 times a second by the processor via a 6520 PIA to check for a key depression. All eight lines on the B port of 6520 number 1 are configured as inputs while lines 0 to 3 of port A are configured as outputs and connected to a four line to ten line decoder. The keyboard is organised in 2 x 5 blocks which are repeated eight times across the keyboard as in Figure 5.3. thus an input line is connected to all the keys in each 2 x 5 block. Key number one in all eight blocks are connected together as are the eight number two keys and so on for all ten keys in each block (note that seven keys are not implemented). The keyboard can be visualised as an eight by ten matrix with eight row lines being connected to the eight inputs on port B and the ten column lines

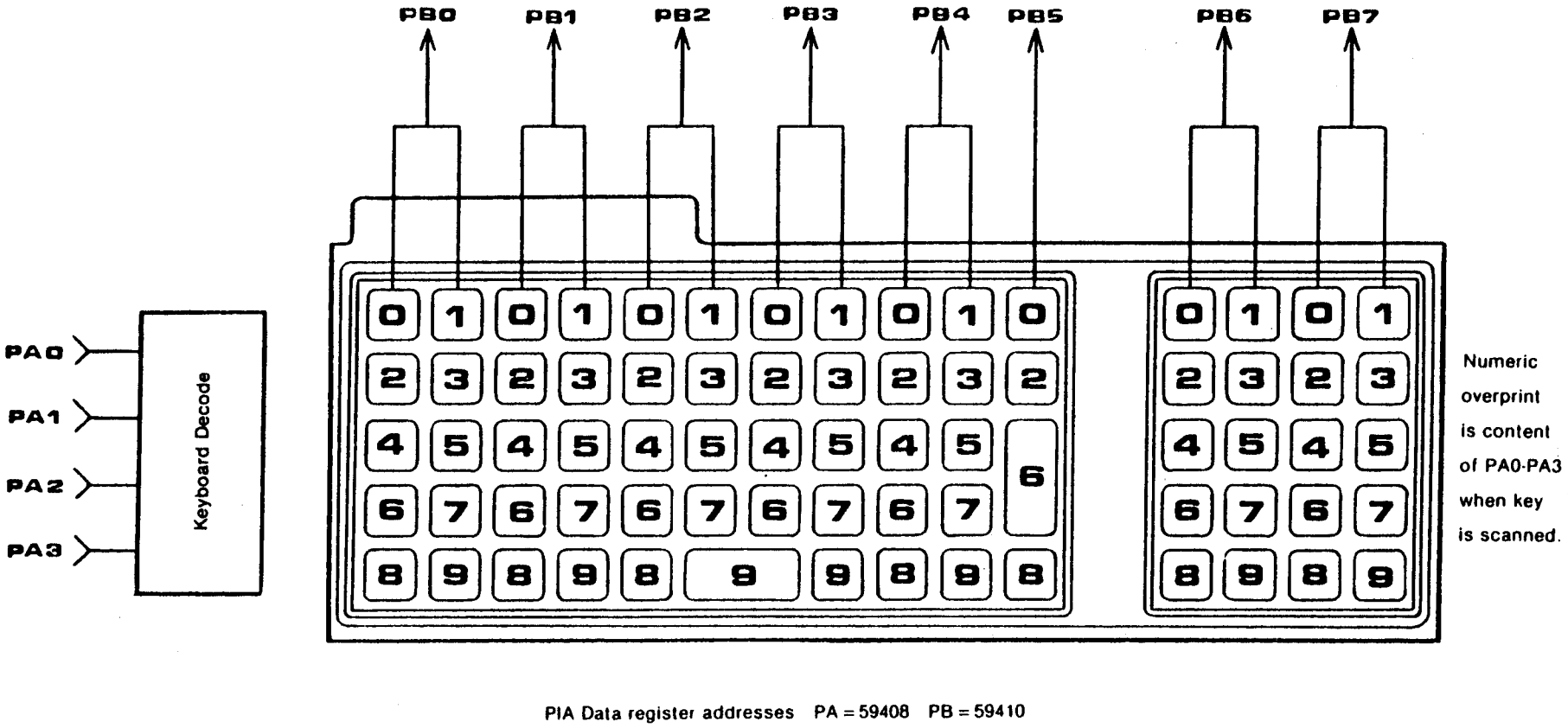


Fig 5.3 PET keyboard layout showing column and row connections.

connected to the ten line decoder output from bits 0 to 3 of port A.

By pressing a key, contact is made between one of the row lines and one of the column lines. If the column line is at a logic 1 then the row line on which the depressed key lies will also be at a logic 1, setting one of the input lines of port B high. If all the column lines were high then a high on one of the row inputs could come from one or all of ten keys being pressed. By having only one column line high at a time an input can come from the depression of only one specific key. The ten column lines are thus scanned by sequentially turning each line on and testing for an input on port B input lines. If an input is found the current column number is recorded together with the input line number for decoding by the operating system. In practice the PET scans a single line at logic zero across the ten column lines which are normally at logic one. The keyboard scanning and decoding subroutine is part of the retrace interrupt initiated once every sixtieth of a second by an interrupt on the CB1 pin of PIA 1. The keyboard can thus be disabled by setting bit zero of the port B control register to 0 which disables the CB1 interrupt. This is very useful since it allows one to protect a program from unauthorised data entry or from being aborted by accidently pressing the stop key. The keyboad can be disabled by the following command from a Basic program: 100 POKE 59411, 60 keyboard function can be enabled again with the command: 100 POKE 59411, 61. The scanning process can best be shown by disabling the retrace interrupt and using a Basic program to perform the same function as the keyboard scanning subroutine in the operating system. The following program while not performing exactly the same function prints out the column and input port B value every time a key depression is sensed:

```
 10 POKE 59411,60 : REM DISABLE KEYBOARD INTERRUPT
 20 FOR Q=1 TO 500: REM DO 50 TIMES
 30 FOR S=0TO9 : REM SCAN COLUMNS 0 TO 9
 40 POKE 59408,S
 50 I=PEEK(59410) : REM LOOK FOR INPUT ON PORT B
 60 IF I < 255 THEN PRINT"INPUT",S,I:S=9:GOTO110
 70 NEXTS
 110 NEXTQ
 120 POKE 59411,61
READY.
```

This program has a major fault, pressing the key for a long time will generate multiple inputs. In a Basic program this is not really a problem since having found

an input, control would normally jump out of the input loop and Basic is too slow for the key still to be pressed on the next scan. However, the operating system scans the keyboard once every sixtyth of a second and it is unlikely that a key depression would be shorter than about 10 keyboard scans. Also when a key is pressed there is bound to be some key bounce which when the keyboard is being scanned could frequently lead to multiple closures being input to the processor. The operating system software is so written that no keyboard scans are accepted until the last key pressed is released. Unless a later scanned key is pressed, this key is then interpreted as being the next key closed even if the first key is still being pressed. To demonstrate: press the Q key then while still pressing Q press the A key. Although the Q key is already pressed an A will be printed on the screen, if the A key is released then another Q will be printed. Protection from noise generated by contact bounce is implemented by the operating system checking that the same key is pressed for more than one scan. At the end of each scan the 6520 is left with column 9 on, this column contains the Stop/Run key which can be tested before doing a full keyboard scan. This is useful since with the keyboard disabled or when operating in machine code the column 9 keys can be used as inputs without having to scan the keyboard. To do this one simply reads I/0 port B of PIA 1 thus: I = PEEK (59410) and if I = 239 then the Stop key has been pressed and if I = 251 then the Space key etc. The subroutine which tests for a depression of the Stop key is located at Hex F8F0 (in old ROMs F32A). The Stop key can be disabled without affecting the rest of the keyboard, this is useful since it allows the programmer to prevent a program being aborted by the user accidently pressing the stop key, whilst still retaining full use of the keyboard. This is done by changing the jump address of the interrupt. The keyboard is scanned by an interrupt service routine the address of which is pointed to by the contents of locations 144 and 145 (old ROMS 537 and 538). The first function of this routine is to test for a depression of the stop key. We can thus disable the stop key by changing the interrupt jump address to point to a location after the stop key detection subroutine call using: POKE 144, ~~228~~ 49 the stop key can be enabled by a POKE 144, ~~225~~ 46.

The only other keys not decoded to give an ASCII character are the two shift keys. The keyboard scanning routine on detecting that either of these keys has been pressed sets a flag in location 152(old ROMs 516). If the decoding subroutine which converts the keyboard matrix co-ordinates into ASCII characters detects that this flag is set, then the program will set bit seven of the associated character thereby converting it to upper case or graphics. It should be noted that two versions

of the ASCII code are used in the PET, one by the operating system and Basic and the other by the video display. It is bit six which is set to give upper case or graphics in the video ASCII code. This knowledge allows one to rectify the slightly annoying feature of the old 8K PET (which has been rectified on the 16 and 32K machines) of having to shift to print lower case, this is done by reversing the contents of bit seven of every character input, thus :

```
 5 POKE59468,14
 10 GETA$:IFA$=""GOTO10
 20 A=ASC(A$)
 30 IFA>128THENB=A-128:GOTO50
 40 B=A+128
 50 A$=CHR$(B)
 60 PRINTA$;
 70 GOTO10
READY.
```

or: To reverse the upper and lower case of all the characters on the screen then one can PEEK the screen contents and reverse bit six, thus:

```
 10 FORI=0TO999
 20 IFPEEK(32768+I)AND64=64THENGOSUB100
 30 GOSUB200
 35 NEXTI
 40 END
 100 POKE32768+I,PEEK(32768+I)AND63
 110 RETURN
 200 POKE32768+I,PEEK(32768+I)OR64
 210 RETURN
READY.
```

The reverse field display key has an associated ASCII character for the reverse "on" mode and another for the reverse "off" mode there is also a reverse field flag in location 159(old ROMs 526). When the RVS key is pressed it is decoded as an ASCII characer with a value of decimal 18, the operating system on recognising this character will set the flag in location 159. This flag is set to indicate to the operating system that all subsequent characters displayed must be reverse field. A character is displayed as a reverse field character if bit seven of the screen ASCII code is set. The reverse field flag in 159 is reset by either a shifted RVS character or by a carriage return. The reverse field "off" character has an ASCII code value of 146. To summarise the display can be put into reverse field by putting an RVS character into the print string or by a CHR$(18); which performs the same funtion. It can also be done by a POKE [illegible],255. All three modes are reset by a carriage return, to reverse the whole screen or a

159

particular section of the screen then one would have to use the following method:

```
10 FOR I = 32768 TO 33769
20 POKE I, PEEK ( I ) OR 128
30 NEXT I
```

The function of all remaining keys is obvious and they are all, including the screen edit keys, decoded to give their own ASCII code. The edit keys are used by the screen edit subroutines of the operating system, allowing the cursor to be moved around the screen under manual or program control. They also allow insertion and deletion of characters or clearing the screen. When used within a string the edit characters are displayed as cryptic graphic characters which can cause a problem when getting a printed listing of a program on a non graphics printer. The ASCII codes can be used to replace the graphics characters producing the same effect, to move the cursor down use: PRINT CHR$ (17) ; the other ASCII codes are as follows:

Cursor up 145	Cursor down 17
Cursor left 157	Cursor right 29
Insert character 148	Delete character 20
Cursor home 19	Screen clear 147
Carriage return 13	

The operating system having performed the keyboard input and character decoding puts the encoded character into a ten character keyboard buffer ready for use by the main program. This buffer is loaded every time a key depression is sensed by the scan subroutine and is unloaded as soon as the characters can be transferred to the screen or to the relevent Basic buffer. The keyboard buffer is organised as a first in first out queue with the address of the last entry being pointed to by the contents of location 158 (old ROMs 525), the buffer is in locations 623 to 632(old ROMs 527 to 536). If the first character in location 623 is taken out all the other characters are moved down one place in the queue, the location pointer in 525 being decremented by one. The keyboard queue can cause problems when running a program since any key pressed before an Input or Get command, will be in the keyboard buffer, giving rise to erroneous inputs. This problem can be overcome by setting the keyboard buffer location pointer to zero just before an Input or Get command this will clear the keyboard buffer of any contents and can be done by a POKE 158,0.

The keyboard buffer can be utilised to create a useful family of programs - programs which actually write their own program lines. This may sound contradictory but there are a great many uses for this

kind of program, perhaps the most useful of these is the automatic writing of Data statements containing values input or calculated by the program itself. The following program will convert a machine code program into Basic data statements.

```
60000 INPUT"[CLEAR]START#,STEP";S,T
60010 INPUT"START ADDRESS DECIMAL";B
60020 F=B:L=F+10
60030 INPUT"END ADDRESS DECIMAL";E
60050 PRINT" CDOWN 4 "
60060 POKE831,INT(E/256)
60070 POKE832,E-INT(E/256)*256
60100 POKE828,T:GOTO60500
60200 S=PEEK(826)*256+PEEK(827)
60300 T=PEEK(828)
60310 L=PEEK(829)*256+PEEK(830)
60330 E=PEEK(831)*256+PEEK(832)
60340 IFL>=EGOTO62000
60350 F=L+1:L=L+10
60400 PRINT"[CUP]                    "
60500 PRINTS;
60600 PRINT"DATA";
60700 FORP=FTOL:PRINTPEEK(P);"[CLEFT],";:NEXTP
60800 PRINT"[CLEFT]  "
60900 PRINT"GOTO60200[CUP 4]";
61000 POKE158,2:POKE623,13:POKE624,13
61100 S=S+T
61200 POKE826,INT(S/256)
61300 POKE827,S-INT(S/256)*256
61400 POKE829,INT(L/256)
61500 POKE830,L-INT(L/256)*256:END
62000 STOP
```

Another use would be in the insertion of algebraic functions into say a graph plotting program, allowing a function in a particular line to be changed either manually or automatically from data without having to use a lot of Gosub and Goto statements. The method is very simple relying on the fact that a line is entered into a program from the screen only after a carriage return is pressed. A program line put on the screen with a print statment can be entered into the main program by clearing the keyboard queue and placing a carriage return into location 623 of the buffer thus :

```
100 PRINT" clear,cdown 3 lines "A$" chome " : POKE 525,1:POKE 527, 13:END
```

Where A$ is the line to be entered into the program or an operation in the immediate mode like GOTO 50 (this

would cause the program to jump to line 50). It should be noted that entering a new line in this manner will destroy all the data and the contents of the subroutine return stack. These values must be stored before this program is executed to be retrieved after execution. This is done in the following example which is an auto line numbering program allowing one to write a program without having to enter the line number for each new line.

```
60000 INPUT"[CLEAR] START#,STEP";S,T
60050 PRINT"[CDOWN 4]"
60100 POKE828,T:GOTO60500
60200 S=PEEK(826)*256+PEEK(827)
60300 T=PEEK(828)
60400 PRINT"[CUP]                    "
60500 PRINTS;
60700 GETD$:IFD$=""THEN60700
60800 PRINTD$;:IFASC(D$)<>13THEN60700
60900 PRINT"GOTO60200[CUP 3]";
61000 POKE158,2:POKE623,13:POKE624,13
61100 S=S+T
61200 POKE826,INT(S/256)
61300 POKE827,S-INT(S/256)*256:END
```

The END command in line 61130 initiates the entry of the new program line, the line number and line increment are stored by poking their values into locations in the second cassette buffer. Line 61091 is an immediate command executed after the program line entry, to return the line numbering program back to 61030 ready for another line entry. If this is incorporated as part of a program then the display on the screen can be disabled (in old static RAM PETs only) by the command POKE 59409,53.

The function of the screen editor subroutine is to transfer the contents of the keyboard buffer to the screen at a position on the screen indicated by the flashing cursor. The editor routines are normally active when no Basic program is running and also during a Basic Input command, in both modes the screen data is entered into the program by a carriage return. Before data is entered into the program it can be edited using the screen edit commands in conjunction with the cursor control command. All line editing is done between the keyboard and the screen memory thereby greatly reducing the complexity of the operating system and the Basic interpreter. The screen editor is not used by the GET command, hence the absence of a cursor during a GET operation. A cursor can however be added to this command in old ROM machines by activating the cursor blink flag

prior to the GET statement with a POKE 548,0. The cursor can also be utilised to prevent the abortion of a program by accidently pressing the return key during an INPUT command. This can be done by formating the Input statement in the following manner:

```
100 INPUT " cright 3 spaces * cleft 3 spaces ";A$
```

This line produces a blinking cursor over an asterisk which disables the stop and return keys, if one of these keys is pressed the command returns with an error message - Redo from start - and a new input prompt. A Keyboard function not implemented on the PET but which the user may like to add is a repeat key, which allows printing of a row of identical characters without having to repeatedly press the same key. Since there are no unused keys on the keyboard one can not have a special key as a repeat key. Instead, holding a key down for a long enough period must be used to generate repeated key presses. To do this one must over-ride the operating system which prevents multiple key closures being registered by inserting extra code into the keyboard scanning interrupt routine.

The program is written in machine code and located in the second cassette buffer. The program consists of two parts, an initialisation routine to enable the repeat key, called by a SYS(832). The second part of the program performs the repeat key function, this tests for a key depression, if found the program delays before repeated characters are generated. Another character is generated by fooling the operating system that the key is not pressed, this is done by writing 255 into location 151 which is the register of the matrix co-ordinates of the last key pressed, a 255 in this location means that no key is pressed. Having generated another character, the program delays before the next repeated character, both delay timings can be varied by changing the relevent values. Once this program has been entered and run it will stay in the machine until the machine is switches off or the program is erased by writing into the second cassette buffer (it should be noted that repeat will affect the operation of both cassettes, IRQ vectors should be re-initialised before using cassettes). The following three programs are first: a machine code listing of repeat for new ROM machines, followed by a Basic loader version of the same program and lastly a Basic loader of repeat for old ROM machines.

```
REPDEL = $02
DELAY = $01
KEY = $00
IRQSUB = $E62E
```

```
                    IRQV = $90
                    LSTKEY = $97
                    BLINK = $A8

                    ;REPEAT KEY ENABLE

0340  78          REPON    SEI
      A9 4F                LDA # REPEAT
      85 90                STA IRQV
      A9 03                LDA # REPEAT +1
      85 91                STA IRQV+1
      A9 01                LDA #1
      85 02                STA REPDEL
      58                   CLI
      60                   RTS

                    ;REPEAT KEY FUNCTION

034F  A5 97       REPEAT   LDA LASTKEY
      C5 00                CMP KEY
      F0 09                BEQ REP1
      85 00                STA KEY
      A9 10                LDA #$10
      85 01                STA DELAY
      4C 2E E6    REPEND   JMP IRQSUB
      C9 FF       REP1     CMP #$FF
      F0 F9                BEQ REPEND
      A5 01                LDA DELAY
      F0 04                BEQ REP2
      C6 01                DEC DELAY
      D0 F1                BNE REPEND
      C6 02       REP2     DEC REPDEL
      D0 ED                BNE REPEND
      A9 04                LDA #$04
      85 02                STA REPDEL
      A9 00                LDA #$00
      85 97                STA LSTKEY
      A9 02                LDA #$02
      85 A8                STA BLINK
      D0 DF                BNE REPEND
```

```
5 REM REPEAT FOR NEW ROM MACHINES
10 FORQ=832TO891
20 READA
30 POKEQ,A
40 NEXTQ
50 STOP
100 DATA120,169,79,133,144,169,3,133,145,169
110 DATA1,133,2,88,96,165,151,197,0,240,9
120 DATA133,0,169,16,133,1,76,46,230,201,255
130 DATA240,249,165,1,240,4,198,1,208,241
140 DATA198,2,208,237,169,4,133,2,169,0,133
150 DATA151,169,2,133,168,208,223
```

```
10 DATA120,56,169,233,237,26,2,141
15 DATA26,2,88,96,173,35,2,201,255
20 DATA208,12,169,0,141,119,3,169
25 DATA90,141,120,3,208,25,238,119
30 DATA3,173,120,3,205,119,3,176,14
35 DATA169,6,141,120,3,162,255,142
40 DATA3,2,232,142,119,3,76,133,230
45 FORI=889TO947
50 READJ
55 POKEI,J
60 NEXTI
```

All devices which the PET communicates with are assigned numbers (except the user port), the keyboard is device 0. This can be used to produce some interesting and useful techniques involving fooling the operating system into thinking that program entry is via the keyboard when in fact it is from another device. These techniques can be used to merge programs together - this method will be looked at in the section on cassette usuage - and inputting programs from another computer connected to the PET via say the IEEE port. This is done by changing the default input device number in location 175 (old ROMs 611). Normally set to 0, the keyboard device number, this location if changed to 1 will fool the system into accepting data from cassette #1 but treating it as if it came from the keyboard. It is however not as simple as poking a 1 into location 175 since the operating system automatically resets this location. Instead one must repeatedly force this input into the PET using the methods already mentioned for automatic line entry. The device number entered into location 175 need not be confined to 1 or 0, it could be 2 if we wanted to input from cassette #2, or 5 to input from a device specified as device 5 on the IEEE port etc.

The Cassette Units.

The standard 8K PET has a single internal cassette unit with the facility of adding another unit via an edge connector at the rear of the machine. New dynamic RAM machines with large keyboards have no internal cassette deck but edge connectors are provided for two external units. The two cassette decks are controlled by I/0 lines from the 6522 VIA and the 6520 PIA #1. Each deck is connected to the PET by six lines - Write, Read, Motor, Sense and two power lines, ground and +5 volts - of these lines only the Write line and the power lines are common between the two cassette units. The connections can be summarised as follows:

	Cassette #1	Cassette #2
Read	CA1 of 6520 #1	CB1 of 6522
Write	PPB3 of 6522	PB3 of 6522
Motor	CB2 of 6520 #1	PB4 of 6522
Sense	PA4 of 6520 #1	PA5 of 6520 #1

The casette motor power supply lines are connected to the interface chips via a three transistor driver used to boost the power and voltage allowing the motor to be driven directly. The output to the motor is an unregulated +9 volts at a power rating of up to 1000ma, (if the second cassette deck is not used this output could be used to power a small external circuit on say the user port). The motor on cassette #1 can thus be turned on and off by toggling the CB2 line on 6520 #1 - POKE 59411,53 should turn the motor on and POKE 59411,61 turn it off, however this will not work unless the scan interrupt is disabled since this automatically turns the motor off.

The sense line input is connected to a switch on the cassette deck which senses when either the Play, Rewind or Fast Forward buttons have been pressed. The switch is only required to sense the pushing of the Play button during a read or write to tape routine this is done by a subroutine at F835 (old ROMs F85E). If either the rewind or fast forward button is pressed accidently instead of the play button the system will be unable to tell the difference and will act as if the play button was pressed. For a similar reason during a record routine the record button must be pressed before the play button since recording will start as soon as the sense switch is closed by pressing the play button.

The functioning of the read and write lines is controlled entirely by the operating system, the only hardware required being signal amplification and pulse shaping circuitry. These circuits are contained on a small PC board within the cassette deck their function being to give correct voltage and current to the record head and amplify the input from the read head to give a 5 volt square wave output able to produce an interrupt on the CA1 or CB1 lines.

In normal usage the two cassette decks are assigned I/O device numbers, the internal cassette is device number 1, the external cassette device number 2. The device number together with the logical file number and the secondary address is used when saving or retrieving data files from one or other of the two cassette decks. The logical file number can be any number from 1 to 255 and is used to allow multiple files to be kept on the same device, it is of little use with casette tape and primarily intended for use with floppy disk units. It is usual to have the logical file number the same as the device number, the logical file number of the current file is stored in location 210(old ROMs 239). The secondary address is important since it determines the operational mode of the cassette, the current secondary address is stored in location 212 and 213 (old ROMs 241 and 242) the normal default value being zero. If the secondary address is zero then the tape is opened for a "read" operation, if set to 1 then it is opened for a

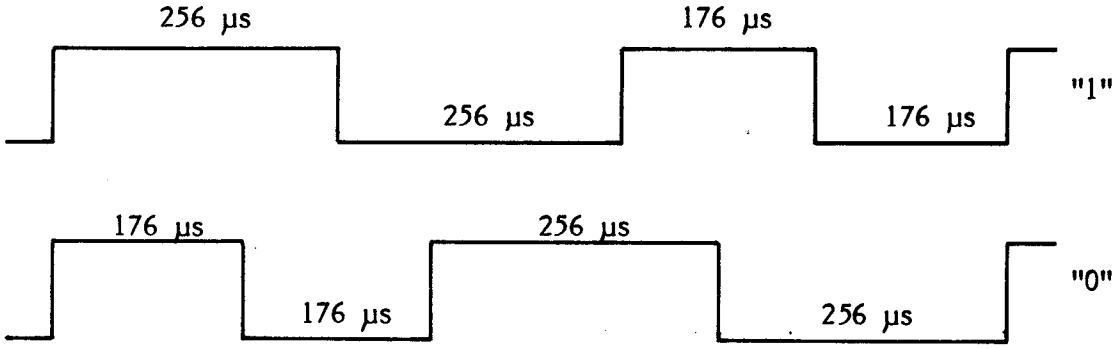
"write" operation and if 2 then it is opened for a "write" with an end of tape header being forced when the file is closed.

The operating system on the PET is configured to allow two different types of file to be stored on cassette: program files and data files. These names are however rather misleading since a program can be stored as a data file and data can be stored as a program file. The difference between these two file types is not in their application but in the way the contents of the machine's memory is recorded. Instead of program and data files we must look upon them as Binary and ASCII files. A binary file is usually used to store programs since a binary file is created by the operating system to store the contents of memory between a starting location and an end location. Called a binary file because the basic statements stored on this file are not stored in the same manner as they are listed on the display or were entered on the keyboard, they are instead stored in the partially encoded form which is used to store the commands within memory. Because the program is stored in a partially encoded form a binary file is a quicker and more efficient way of storing programs, and essential if saving and loading machine code programs and data. The starting address from which a binary file will be saved is stored in locations 251 and 252 (old ROMs 247 and 248), normally these will be set to 0 and 4 thereby pointing to the start of the Basic text area at 1024. They can be altered to point to any location in memory. The end address of the area of memory to be saved is stored in locations 201 and 202 (old ROMs 229 and 230) normally when saving a Basic program these are set to the last address of the last statement. Like the beginning, the end address can be altered to any desired address. To change either of these addresses one can not use the normal save routine since this automatically initialises these locations. Instead one must write a small machine code initialisation routine incorporating the desired operating system subroutines (see No copy program). By default a Save command will write a binary file and a Load command will read a binary file.

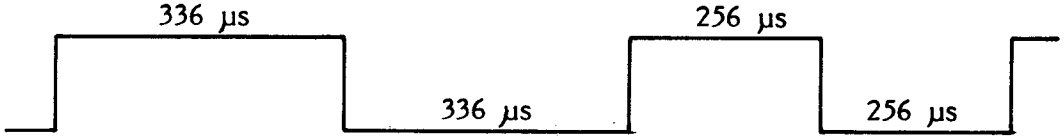
An ASCII file is normally used to store data (but can be used to store programs see Merge procedure) the format being the same as that displayed on the screen or entered on the keyboard. ASCII files are created or read almost exclusively by instructions from within a Basic program. A binary file is created or read exclusively by direct instructions, though the Load and Save instructions can be used within a program. An ASCII file must first be opened with an Open statement which specifies the logical file, device number, secondary address and file name. This is then interpreted by the operating system allowing the user to read or write the

file to the specified device. Data is written to an ASCII file on a particular device with a command to Print to the specified logical file number, and data is read by a Read from logical file command. Whereas a binary file is loaded with the contents of successive memory locations, an ASCII file is loaded with a string of variables. Storing these individually would require the tape to be turned on and off repeatedly storing a few bytes of data at a time. The PET overcomes this by having a 192 byte tape buffer for each cassette deck into which all data to be written to, or read from tape is loaded, only when this buffer is full is the tape motor turned on. Data is stored on tape in blocks of 192 bytes and since the motor is turned on and off between blocks a two second interval must be left between blocks to allow the motor to accelerate and decelerate. The beginning of the 192 character buffer for cassette #1 starts at address 634 and for cassette #2 at location 826. The pointer to the start of these buffers is located at address 214 and 215(old ROMs 243 and 244). The number of characters in a buffer is stored in locations 187 for buffer #1 and 188 for buffer #2 (old ROMs 625 and 626), these locations can be used by the programmer to control the amount of space left in a data file. If having opened a file on cassette #1 the command POKE 625,191 is executed then the contents of the tape buffer even if empty is loaded onto the tape. If records are kept in multiples of 191 bytes we can very easily keep nul or partially filled records allowing future data expansion.

Whether the file being stored is binary or ASCII the recording method used is the same involving an encoding method unique to Commodore and designed to ensure maximum reliability of recording and playback. Each byte of data or program is encoded by the operating system using pulses of three distinct audio frequencies, these are: long pulses with a frequency of 1488Hz, medium pulses at 1953Hz and short pulses at 2840Hz. All these pulses are square waves with a mark space ratio of 1:1, one cycle of a medium frequency is 256 microseconds in the high state and 256 microseconds in the low state. The operating system takes about 9 milliseconds to record a byte of data consisting of the eight data bits, a word marker bit and an odd parity bit. The databits are either ones or zeros and are encoded by a sequence of medium and short pulses: a "1" is one cycle of a medium length pulse followed by one cycle of a short length pulse and "0" is one cycle of a short length pulse followed by one cycle of a medium length pulse. Each bit consists of two square wave pulse cycles, one short and one medium with a total duration of 864 microseconds as in the following diagram:



The odd parity bit is required for error checking and is similarly encoded, its state being determined by the contents of eight data bits. The word marker is used to separate each byte of data and also to signal to the operating system the beginning of each byte. The word marker is encoded as one cycle of a long pulse followed by one cycle of a medium pulse thus :



Since a byte of data is recorded in just 8.96 milliseconds a 191 byte block of data in an ASCII file should be recorded in just over 1.7 seconds, however on timing such a recording we find it takes 5.7 seconds. There are two causes for this discrepency in timing, firstly to reduce the possibility of audio dropouts the data is recorded twice, secondly a two second interrecord gap is left between each record of 192 bytes. The extensive use of error checking techniques is one reason why the tape system on the PET is so much better than that available on most other popular computers. There are two levels of error checking, the first divides the data into blocks of eight bytes and then computes a ninth byte which is a checksum digit, this is obtained by adding the eight bytes together and taking the least significant byte of the result. If when the tape is read one bit in the eight bytes is dropped and a zero becomes a one and the same procedure is applied to calculate the check digit, the result will be different to that stored in byte nine, the check digit of that block computed when the tape was recorded. The second level of error checking involves recording each block of data twice and if an error was detected by the check digit performing a verification process between the two blocks.

The use of pulse sequences rather than two frequencies as in a standard FSK recording has a great

advantage since it allows the operating system to easily compensate for variations in recording speed. Normally a hardware phase locked loop circuit would be used to lock the system onto the correct frequencies coming from the tape head, the PET however uses software to perform this process. A ten second leader is written on the tape before recording of the data or program commences. This leader has two functions, first it allows the tape motor to reach the correct speed and secondly the sequence of short pulses written on the leader is used to synchronise the read routine timing to the timing on the tape. The operating system can thus produce a correction factor which allows a very wide variation in tape speed without affecting reading. The system timing used to perform both reading and writing is very accurate, based as it is on the crystal controlled system clock via the internal timer #1 on the 6522 chip. Interrecord gaps are only used in ASCII files and their function is to allow the tape motor time to decelerate after being turned off and accelerate to the correct speed when turned on prior to a block read or write. Each interrecord gap is approximately two seconds long and is recorded as a sequence of short pulses in the same manner as the ten second leader. There is also a gap between blocks, when the first block of 192 bytes is recorded it is followed by a block end marker which consists of one single long pulse followed by 50+ cycles of short pulses then the second recording of the 192 block starts, this is identical to the first block.

The first record written on the tape after the ten second leader in both ASCII and binary files is a 192 character file header block. The file header contains the name of the file, the starting memory location, and the end location. In an ASCII file these addresses are the beginning and end of the tape buffer, in a binary file they point to the area of memory in which the program is to be stored.

The file name can be up to 128 bytes long, the length of the file name is stored in location 209(old ROMs 238), and when read is compared with the requested file name in the Load or Open command. If the name is the same then the operating system will read the file, if different then it will search for the next ten second interfile gap and another header block. The file name is stored during a read or write operation in a block memory, the starting address of which is stored locations 218 and 219 (old ROMs 249 and 250), on completion of the operation these are reset to point to a location in the operating system. The starting location is normally set to the beginning of the user memory area, address 1024, however it can be changed to point to any location, a method employed when recording programs in machine code using the monitor,and also in the Nocopy program shown later in this chapter. The

starting address is pointed to by the contents of locations 251 and 252 (old ROMs 247 and 248). The end address being stored in locations 201 and 202 (old ROMs 229 and 230) normally this is the highest byte of memory occupied by the program, however it can be altered to point to any address providing it is greater than the start address.

Normally any program running on the PET whether in Basic or machine code can be saved on tape, this fact has deterred many programmers from writing quality commercial software for the machine since it is so easy to make a copy. However machine code programs can be made uncopyable by using a special save routine, the program when recorded changes the file header contents in such a way that prevents any further copies of the tape being made. The program works by setting the start address to a location just below the user memory area, instead of 1024 locations 251 and 252 now contain 1021 so that the program starts at this address. If we try running a program from this location we will simply get an out of memory error since the operating system now looks upon location 1024 as being the highest memory location. To overcome this a jump instruction -Hex 4C- is put at address 1024. When the program is run it works perfectly normally, however, when an attempt is made to save the program the machine will respond with an out of memory error. The start location can be lower than 1021, this allows the second cassette buffer to be used as well as the main memory. The following program will create a binary tape of the entire memory contents from location 826 to 8192 and gives it the file name "SAVE", the locations and file name can be changed by the user by changing the relevant locations.

```
033A    A9 4C       LDA 4C
        8D FD 03    STA 03FD      :store jump instruction
                                  in 1021
        A9 01       LDA 01
        85 D4       STA Z D4      :current secondary address
                                  in 212
        A9 67       LDA 67
        85 DA       STA Z DA      :LSB of ile name location
                                  in 218
        A9 03       LDA 03
        85 DB       STA Z DB      :MSB of file name location
                                  in 219
        A9 02       LDA 02
        85 D1       STA Z D1      :file name length in
                                  location 209
        A9 3A       LDA 3A
        85 FB       STA Z FB      :LSB of start address
                                  in location 251
        A9 03       LDA 03
        85 FC       STA Z FC      :MSB of start address
                                  in location 252
```

```
A9 00        LDA 00
85 C9        STA Z C9      :LSB of end address in
                           location 201
A9 20        LDA 20
85 CA        STA Z CA      :MSB of end address in
                           location 202
A2 00        LDX 00
20 9E F6     JSR F69E      :jump into "Save" subroutine
4C 8B C3     JMP C38B      :jump to "Ready" subroutine
53           BYT           :S - first character of
                          file name
41           BYT           :A
56           BYT           :V
45           BYT           :E
```

Since this program is of use only with machine code programs the Nocopy program is best entered and saved using the machine code monitor. To demonstrate its function, use the monitor to enter the program and then save the monitor and the Nocopy program with a SYS(826) from the Basic mode. Switch the PET off and reload using the new tape, you will find it impossible to make a copy of this new tape in the conventional manner, further copies can only be made by the Nocopy program.

Whenever a Basic program is loaded into the PET it will always start at location 1024 meaning that we can not merge programs together since if we load another program it will simply overlay the first program. The secret of merging two programs is, having loaded a program new lines can be entered from the keyboard and existing lines amended. By changing a few locations we can fool the operating system into accepting data from the cassette as if it were the keyboard. This requires that the subroutines or program which we want to merge into our main program are stored as ASCII tapes rather then the normal binary tape. The reason being that the contents of the tape must be the same sequence of characters entered on the keyboard and not the compressed form stored on a binary tape. A program can be easily saved as an ASCII tape by using the following sequence of commands:

OPEN 1,1,1 : CMD 1 : LIST

This lists the program to cassette #1 rather than the screen or a printer, when the program has been recorded the PET can be returned to normal operation by the command:

PRINT #1 : CLOSE 1

Using this process one can build up a library of useful

and/or common subroutines, however, one must be careful to number the lines according to some method whereby subroutines are divided into groups each with its own unique block of line numbers. The reason being that using this merge routine subroutine line numbers which are the same as line numbers in the main program will erase the main program lines. Also if the line number of the subroutine and the main program overlap even though none of them have the same number the subroutine lines will be inserted between those of the main program. Another point to watch is the use of variable names in a subroutine, these should conform to a standard where a particular variable name is always used exclusively to perform a particular function in all subroutines and programs. This helps to avoid the confusion which can result from using the same variable for two purposes.

The process of merging a subroutine stored as an ASCII file into a main program stored in the PETs memory is quite simple but must be done exactly as follows otherwise the process will not work. The first step is to insert the subroutine program tape, rewind and type :

```
OPEN 1
```

The Pet will respond with a prompt to press the PLAY button on the cassette, do this and then wait for the tape to stop. In my experience there are times when the tape deck motor does not stop after ten or fifteen seconds as it should , in this case press the Stop key, rewind, and repeat the above process. By opening the file in this manner the operating system reads the tape header and initialises the system to read data from the tape. Then it stops the tape in the interrecord gap prior to the first 192 byte record. For the processor to read this record and interpret it as program lines entered on the keyboard, requires a little trick incorporating the methods used for automatic line entry. Before a record can be read the default input device number in location 175 (old ROMs 611) must be changed from 0, this is the keyboard, to 1 which is the device number for cassette #1. This can not be done by a POKE 175,1 the system will crash by responding with READY then SYNTAX ERROR then PRESS PLAY ON TAPE 3. One must catch the system between the ready response and the syntax error and enter another POKE 175,1 thereby maintaining the stability of the system with a device number of 1. This is possible by forcing a carriage return into the keyboard buffer and moving the cursor back to the "home" position, when the processor responds by printing READY the cursor is placed on the beginning of the line containing the POKE 175,1 command which it then executes again. To do this the screen is cleared, the cursor moved down four lines and the following line entered:

POKE 175,1:POKE 158,1: POKE 623,13:?" home cursor "

The reason for moving the cursor down four lines is to provide space for the READY response to be printed. Instead of pressing return after entering this line press "cursor home", then move the cursor down six lines and enter the same line again. Make sure the play button on the cassette is still down then press return, the tape should move and the subroutine entered. This line is entered twice so that when the line which has just been processed is four lines down from the top it will automatically execute another line which is six lines down from the top of the screen and vice versa. For this reason two identical lines must be put on the screen one on line four the other on line six. When the merge is completed the message ?SYNTAX or ?OUT OF DATA will be printed on line five and the tape should stop if not then press the RUN/STOP key. Normal operation of the PET can be resumed by closing the file with the command:

CLOSE 1

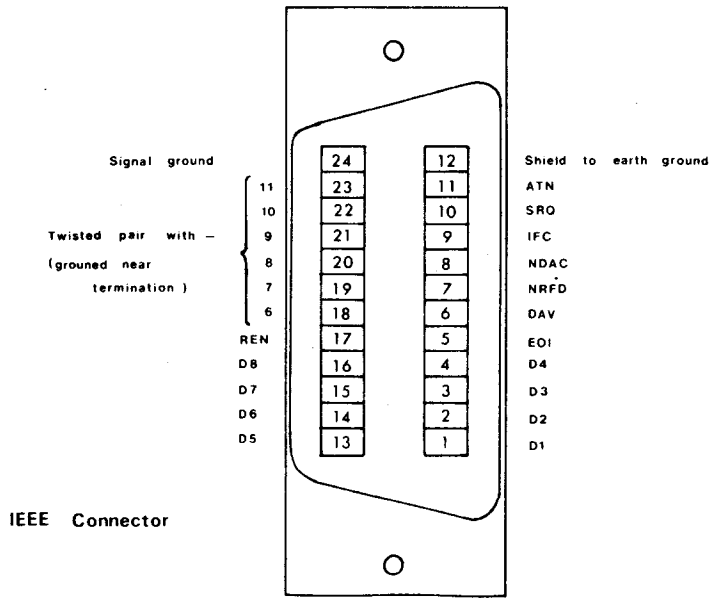
On listing the program you should find that the subroutine has been inserted into the correct position in the main program.

The IEEE Port.

The IEEE-488 port is the principle I/0 port on the PET, designed to allow the PET to be connected to a wide range of peripheral devices ranging from printers and the PET floppy disk to scientific instruments. The IEEE-488 bus or as it is sometimes known the HP-IB bus was developed by Hewlett Packard in the early 1970s to simplify the integration of instruments, calculators and computers into systems. It has since been adopted as an international standard bus, the standards being laid down by the American Institute of Electrical and Electronic Engineers and given the standard number 488. This means that it should be possible to connect any IEEE 488 device to any other IEEE 488 device. This has prompted many manufacturers throughout the world to produce equipment with IEEE 488 interfaces. This fact coupled with a belief that the IEEE 488 bus will become the only standard way of interfacing computers and peripherals prompted Commodore to use this bus on the PET in preference to say an RS232 I/0 port. The use of an IEEE 488 port on the PET has met a mixed reception (some claiming Commodore's decision to use it, a stroke of genius, others claiming it a disaster). However, it is not hard to construct an IEEE 488 to RS232 interface and in this way the PET user can have the best of both worlds.

The sixteen active lines of the IEEE port are

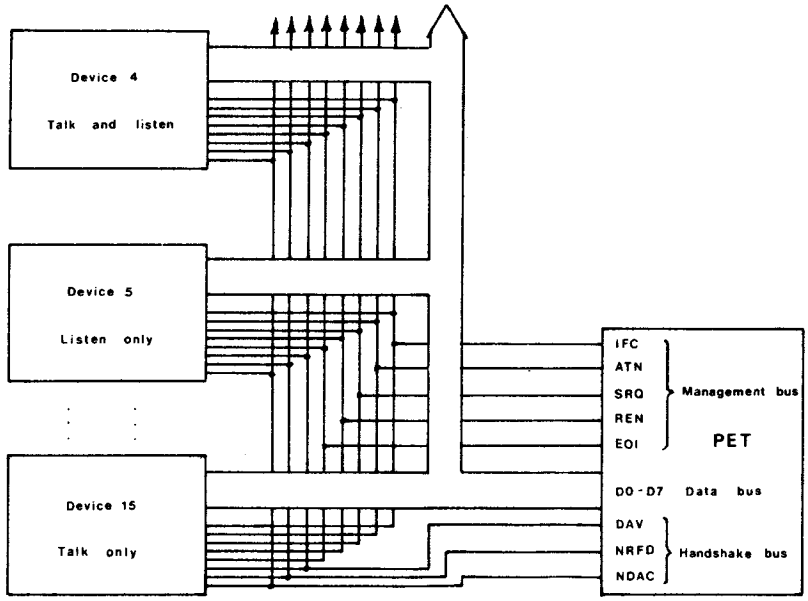
principally derived from 6520 #2. Only four lines are connected directly to the interface chips or the processor control bus, the remainder being connected to the system via three quad line bi-directional buffer ICs. The bi-directional buffers are used to combine two lines, one input and one output, from the peripheral I/0 chips to produce the bidirectional lines required by the IEEE bus. From the processors view the IEEE port consists of eight data input lines and eight data output lines plus four handshake outputs and four handshake inputs, the remaining four control lines are unidirectional. The bi-directional buffer chips are tri-state devices, in the non active state the bi-directional lines on the IEEE port are at a high impedance state. This means that they have a voltage level intermediate between the high state and a low state allowing any device to hold the bus in a "true" or logical "1" state. The standard IEEE connector is not used on the PET, instead as with other I/0 port connectors it is a 12 position 24 contact edge connector with a .156 inch space between the contact centres. If the IEEE port is to be used with instrumentation then the user must add a standard connector which is a 24 contact type 57 Microribbon connector the connections for which are shown in the following diagram.



IEEE Connector

The maximum length of cable used to connect devices together on the IEEE bus should not exceed more than 5 metres and the length of the cable between the PET and the last device on the bus should not exceed 15 metres. A great virtue of the IEEE port is that one can use it to connect more than one device to the computer hence

the reason why it is often referred to as the IEEE bus. Each device is identified on the bus by its device number, the PET allows the user to connect up to 15 different devices onto the IEEE bus. An example of the way such devices are connected onto the PET IEEE bus is shown in the following diagram. Each device is connected in parallel to the 16 lines of the bus these sixteen lines being the sole communication link between the devices and the PET controller.



The devices connected onto the IEEE bus must be capable of performing at least one of the following functions:

LISTENER - A device which is defined as a listener must be capable of receiving data from other devices connected to the bus. The best example of a device which acts solely as a listener is a printer.

TALKER - A device capable of transmitting data to other devices on the IEEE bus. An example of this is a digital voltmeter, others would be a counter or a paper tape reader.

CONTROLLER - A device which manages the communications over the IEEE bus such as addressing devices and sending commands. The PET is the only device which can act as a controller, the controller of course can also act as either a talker or listener.

Although up to 15 devices can be put onto the IEEE

bus only one device at a time can act as a talker, all other devices can simultaineously act as listeners allowing data to be input to more than one device at a time.

The sixteen signal lines of the IEEE bus can be divided into three groups, these are: the data transmission bus, the transfer bus and the management bus; the remaining eight lines on the 24 line connector are grounds. The data bus consists of eight bi-directional lines for transmission of data signals in a bit parallel mode, the signals are active low and the most significant bit is on line D108. The data is transmitted one byte at a time as a seven bit ASCII code with the eighth bit available for a parity check, the data transmission rate is controlled by the slowest device on the bus at a particular time. Although the maximum data transfer rate on the IEEE bus is about 1M bytes per second the PET is limited by the processor speed, practical limits are about 5000 bytes per second, in Basic this is reduced to 100 bytes per second. The data bus is also used to transmit peripheral addresses, these are device addresses used to enable a device to be accessed on the bus. Also control information, both are distinguished from data by having the ATN line low during transfer. The transfer bus consists of three lines used to control the transfer of data over the data bus, as with the data lines these signals are active low. The function of the transfer bus lines can be summarised as follows:

DAV Data Valid

When this line is low it signals that there is valid data on the data bus.

NRFD Not ready for data

This line is kept low for as long as one or more devices on the IEEE bus defined as listeners are not ready to accept data. As soon as all devices are ready NRFD goes high

NDAC Data not accepted

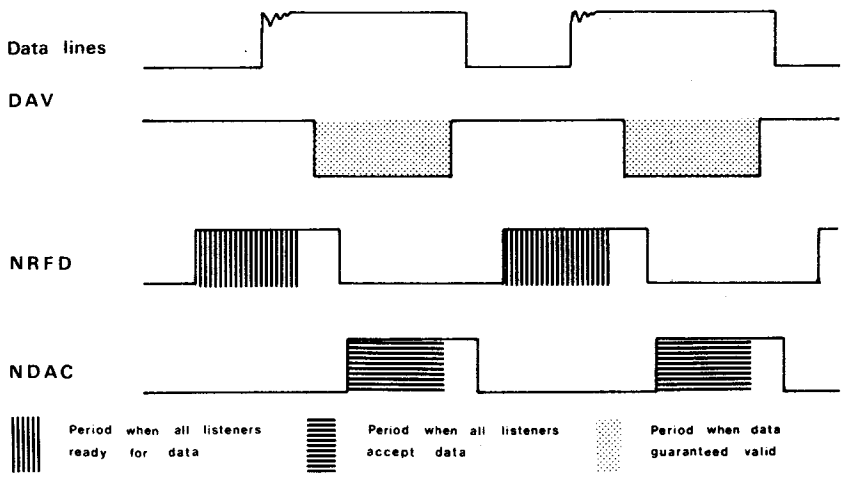
This line is held low by a listening device while reading data, as soon as the data has been read the listener sets NDAC high thus signalling to the talker that the data has been accepted.

Since data is transferred on the IEEE bus in an

PET contact label	Bus	IEEE label	PET contact number	Description
1	DATA	D101	1	Data INPUT/OUTPUT LINE #1
2		D102	2	Data INPUT/OUTPUT LINE #2
3		D103	3	Data NPUT/OUTPUT LINE #3
4		D104	4	Data INPUT/OUTPUT LINE #4
5	MANAGER	EO1	5	End of identify
6	TRANSFER	DAV	6	Data valid
7		NRFD	7	Not eady for data
8		NDAC	8	Data not accepted
9	MANAGER	C		I terface Same as PET reset
10		SRQ	10	Service request
11		ATN	11	Attention
12		SHIELD	12	Chassis ground and IEEE cable shield
A	DATA	D105	13	Data INPUT/OUTPUT LINE #5
B		D106	14	Data INPUT/OUTPUT LINE #6
C		D107	15	Data INPUT/OUTPUT LINE #7
D		D108	16	Data INPUT/OUTPUT LINE #8
E	MANAGER	REN	17	Remote enable (REN) always ground in the PET
F	GROUNDS	GND6	18	DAV ground
H		GND7	19	NFRD ground
J		GND8	20	NDAC ground
K		GND9	21	IFC ground
L		GND10	22	SRQ ground
M		GND11	23	ATN ground
N		LOGIC GND	24	Data ground (D101.8)

Fig 5.4 IEEE Port Connections

asynchronous mode the function of the three lines of the transfer bus is to handshake data transfers between a talker and a listener. The timing of the handshaking sequence is very important and is best illustrated by showing the actual waveforms of the three transfer bus lines and the data bus lines over two cycles of a handshaking sequence where two bytes of data are transferred from one talker to one or more listeners.



With the PET there are some constraints on the timing of the handshaking sequence which must be observed if loss of data is to be avoided, these are:
1) when the PET is a listener the DAV line must go low within 64 milliseconds after it has set NFRD high.
2) when the PET is a talker then NADC must go high within 64 milliseconds after it has set NRFD high.
The five lines which comprise the management bus are to give device commands and to control the current state of the data bus the functions of these lins can be summarised as follows:

ATN Attention

This line is set low by the controller when it is sending commands and peripheral addresses on the data bus. As soon as ATN goes high the previously assigned devices can transfer data between themselves and the controller.

EOI End of Identify

This line is set low by the talker while the last byte of data is being transfered and thus indicates to the listener that it is the end of the message.

IFC Interface Clear

The IFC line on the PET is connected to the systems reset, thus when the PET is switched on this line goes low for about 100 milliseconds. By setting the IFC line low all devices connected to the IEEE bus are initialised to an idle state.

SRQ Service Request

Some devices connected to the IEEE bus have the ability to request service from the controller and it does this by setting the SRQ line low. This line however is not implemented by Basic on the PET but it is connected to the CB1 input on 6520 #2 and can be used by writing a machine code subroutine to test the state of this line as part of the 60Hz keyboard scan interrupt. If more than one device can set the SRQ line low then the controller must poll the devices to find which one requested service, the controller does this by transmitting the serial poll mode command which is hex 18. Each device is then polled by setting ATN, addressing the device as a talker and then removing ATN, if it was that device which set SRQ then it will respond by setting data line 7 low. The serial poll mode is disabled by the controller transmitting command hex 19.

REN Remote Enable

This line is held low by the PET and is not under user control

The PET as the only active controller allowed on the bus manages all communications between devices, doing this by sending commands to these devices via the data lines. Commands are distinguished from data by the state of the ATN line, when this is low the data bus is in the command mode and the controller the only active device, all other devices are waiting for instructions. These commands are performed automatically by the operating system of the PET when the IEEE bus is being used under Basic. A knowledge of the commands is required if the bus is to be controlled under machine code.

The simplest group of commands are address and unaddress, there are four of these commands: talker address, listener address, unlisten address and untalk address. The talker address is transmitted as a seven bit code and enables a specific device to talk, since only one device at a time can act as a talker this command automatically unaddresses and disables the previous talker. The talker address is functionally the

same as the device number used in Basic but whereas a device number can be any number from 4 to 30, the talker address is any one of a group of 31 seven bit byte ASCII characters which are defined as talk addresses by bit six = 0 and bit seven = 1. Each device has its own unique talk address which can be set by the user and will be used by the controller software to select that device. The listener address is also transmitted as a seven bit code used to enable a specific device to act as a listener. A listener address is the same as the device number used in Basic and can be any one of a group of 31 seven bit byte ASCII characters defined as listener addresses by having bit six = 1 and bit seven = 0. Note that in Basic the difference between a talker and a listener is determined by the contents of the secondary address which the operating system translates into the values of bit six and seven. When a device can act as both talker and listener then they are assigned addresses which are identical except for the contents of bits six and seven. A device selected as a listener by the ASCII character "&" has a talker address selected by the character "F". Both talker and listener addresses can be changed by the user, this is normally done by adjusting a set of switches or jumpers within the instrument.

A device selected by a listen or talk address can be deselected by an unaddress command. The unlisten command which is hexadecimal 3F clears the bus of all listeners. The untalk command which is hexadecimal 5F disables the current talker so that no talker remains on the bus, this effect can also be achieved by selecting an unused address.

A device need not be addressed to respond to a set of commands known as universal commands, and all devices on the bus will respond to one of these commands from the controller irrespective of whether they are addressed or not. There are five universal commands and their functions are summarised as follows:

DCL Device Clear Hex 14

This command returns all devices on the IEEE bus capable of responding to a predetermined state irrespective of whether they are addressed or not.

SPE Serial Poll Enable Hex 18

This enables the serial poll mode on the bus, it is only used when the SRQ line is implemented on the PET, this mode enables the controller to find which device generated the service request.

SPD Serial Poll Disable Hex 19

The serial poll mode set by the SPE command is disabled by this command.

LLO Local LOckout Hex 11

The local reset button on the front panel of a responding device can be disabled by this command.

PPU Parallel Poll Unconfigured Hex 15

This provides all devices on the IEEE bus capable of responding to this command with the ability to uniquely identify itself if it requires service and the controller is requesting a response. This command differs from service request since it requires the controller to periodically conduct a parallel poll. This command is not implemented on the PET by Basic.

The remaining set of IEEE commands are all addressed commands and affect only those devices which have previously been defined as listeners. The virtue of addressed commands is that they allow the controller to initiate an action in either a single instrument or a simultaneous action in a group of instruments. There are five addressed commands and their functions can be summarised as follows:

SDC Selective Device Clear Hex 04

This command returns all addressed devices on the IEEE bus capable of responding to a predetermined state.

GTL Go to Local Hex 01

Returns the addressed devices to local control.

GET Group Execute Trigger Hex 08

Initiates a simultaineous pre-programmed action by a group of addressed devices.

PPC Parallel Poll Configure Hex 05

This performs a similar function to the parallel poll unconfigured command, it permits a single DI0 line to be

assigned to each instrument (maximum number of devices is thus eight) for the purpose of responding to the parallel poll.

TCT Take Control Hex 09

This command alows the active controller of the IEEE bus to transfer control to another device. This can not be implemented on the PET since the operating system only allows the PET to act as the active controller.

Other commands specific to a particular device can be given on the IEEE bus, these are the secondary address commands used in the OPEN statement to instruct an inteligent peripheral to function in one of a number of different modes. The form and nature of a secondary address command whether given from Basic or machine code depends entirely on the device. Each device has its own conventions which can only be obtained by consulting the manual for the device. A secondary address can have a value between 0 and 31 in Basic. Note that when the Basic secondary address is transmitted it is as the OR of hex FO since bits 4,5,6 and 7 must be set.

The OPEN command in Basic is used to select a device on the IEEE bus which has a device number between 4 and 30. If the device number is less than 4 the operating system will instead address either the keyboard, cassettes or screen. The operating system is also initialised so that the device will communicate with a particular logical file having a number between 1 and 255. The use of a secondary address and a file name is optional, however, a secondary address is only sent if a file name is used, the operating system then sends a listen command to the specified device followed by the secondary address. If there is no response by the device to the ATN command the operating system will respond with a "DEVICE NOT PRESENT" error and set bit 7 of the status byte. Having initialised the system and a specified device for data transfer on the IEEE bus and perhaps set the addressed device to a particular function by using the secondary address, data can be transfered using either the INPUT #, PRINT# or GET # commands. When one of these commands is encountered in a program the operating system will go through the IEEE-488 input initiation routine. The INPUT# and GET # commands specify a particular logical file number, the input initiation routine sends a talk command to the device specified in the OPEN command for that logical file, setting the addressed device as a talker and the PET as a listener. The PET then waits for the DAV line to be set low indicating that the talker has placed a single byte of valid data on the bus. An input on the

Talk Addresses								
Bits								ASCII
b8	b7	d6	b5	b4	b3	b2	b1	Character
X	1	0	0	0	0	0	0	@
X	1	0	0	0	0	0	1	A
X	1	0	0	0	0	1	0	B
X	1	0	0	0	0	1	1	C
X	1	0	0	0	1	0	0	D
X	1	0	0	0	1	0	1	E
X	1	0	0	0	1	1	0	F
X	1	0	0	0	1	1	1	G
X	1	0	0	1	0	0	0	H
X	1	0	0	1	0	0	1	I
X	1	0	0	1	0	1	0	J
X	1	0	0	1	0	1	1	K
X	1	0	0	1	1	0	0	L
X	1	0	0	1	1	0	1	M
X	1	0	0	1	1	1	0	N
X	1	0	0	1	1	1	1	O
X	1	0	1	0	0	0	0	P
X	1	0	1	0	0	0	1	Q
X	1	0	1	0	0	1	0	R
X	1	0	1	0	0	1	1	S
X	1	0	1	0	1	0	0	T
X	1	0	1	0	1	0	1	U
X	1	0	1	0	1	1	0	V
X	1	0	1	0	1	1	1	W
X	1	0	1	1	0	0	0	X
X	1	0	1	1	0	0	1	Y
X	1	0	1	1	0	1	0	Z
X	1	0	1	1	0	1	1	
X	1	0	1	1	1	0	0	
X	1	0	1	1	1	0	1	
X	1	0	1	1	1	1	0	

Table of IEEE Device Talk Addresses

Listen Addresses								
Bits								ASCII
b8	b7	b6	b5	b4	b3	b2	b1	Character
X	0	1	0	0	0	0	0	SP
X	0	1	0	0	0	0	1	!
X	0	1	0	0	0	1	0	"
X	0	1	0	0	0	1	1	#
X	0	1	0	0	1	0	0	S
X	0	1	0	0	1	0	1	%
X	0	1	0	0	1	1	0	&
X	0	1	0	0	1	1	1	'
X	0	1	0	1	0	0	0	(
X	0	1	0	1	0	0	1	)
X	0	1	0	1	0	1	0	*
X	0	1	0	1	0	1	1	+
X	0	1	0	1	1	0	0	
X	0	1	0	1	1	0	1	-
X	0	1	0	1	1	1	0	.
X	0	1	0	1	1	1	1	/
X	0	1	1	0	0	0	0	0
X	0	1	1	0	0	0	1	1
X	0	1	1	0	0	1	0	2
X	0	1	1	0	0	1	1	3
X	0	1	1	0	1	0	0	4
X	0	1	1	0	1	0	1	5
X	0	1	1	0	1	1	0	6
X	0	1	1	0	1	1	1	7
X	0	1	1	1	0	0	0	8
X	0	1	1	1	0	0	1	9
X	0	1	1	1	0	1	0	:
X	0	1	1	1	0	1	1	;
X	0	1	1	1	1	0	0	
X	0	1	1	1	1	0	1	=
X	0	1	1	1	1	1	0	

X = don't care

Table of IEEE Device Listen Addresses

DAV line must be received within 64 milliseconds if that byte of data is to be placed in the Basic input buffer. If not received within that period then the IEEE input sequence will be terminated and the error handling routine will set the status byte in Basic variable ST to 2, indicating a talker time out. The status byte is stored in location 150 (old ROMs 524) and the setting of bit 1 by a time out error can be used to prevent the program returning to command mode after the error. This is done by following the INPUT # or GET # command immediately with a test of the status byte and if bit 1 is set then control returns to the INPUT or GET command, thus:

```
100 INPUT # 5,5,2,"A"
110 IF ST =0 THEN 120: GOTO 100
120....
```

If the Basic command was INPUT # then having fetched one character and placed it in the input buffer, the IEEE input routine is called again and another character input. This process is continued until the input routine senses a low level on the EOI line which indicates the end of information transfer. Note: not all devices generate an EOI signal. On sensing an EOI pulse the operating system will set bit six of the status byte and will force carriage return into the buffer until the current command is terminated. The INPUT # command is limited by the length of the input buffer which prevents the transfer of more than 80 characters at a time unless a carriage return separates each 80 character block. Any attempt to write more than 80 characters into the buffer which is located between locations 512 and 591 (old ROMs 10-89) will result in system malfunction. If the IEEE device sends more than 80 characters without a carriage return between blocks, then the GET command must be used, since this command only calls the IEEE input routine once and thus only inputs one character each time the command is executed. By repeatedly performing the GET # command strings of data can be built up which avoid the buffer size limitations but are unfortunately rather slow. At the end of an input command whether it was INPUT # or GET # an IEEE termination routine is called which returns the default input device number in location 175 (old ROMs 611) to 0 thereby restoring the functioning of the keyboard. An untalk command is then set to the IEEE bus freeing it for the next command.

Having opened a logical file to a specific device the PET can output data to that device with a PRINT # command which calls an IEEE output subroutine. This sets the device specified for the logical file in the PRINT # command into a listener mode. The operating system then changes the default output device number in 176 from 3 which is the video display to the device which has just

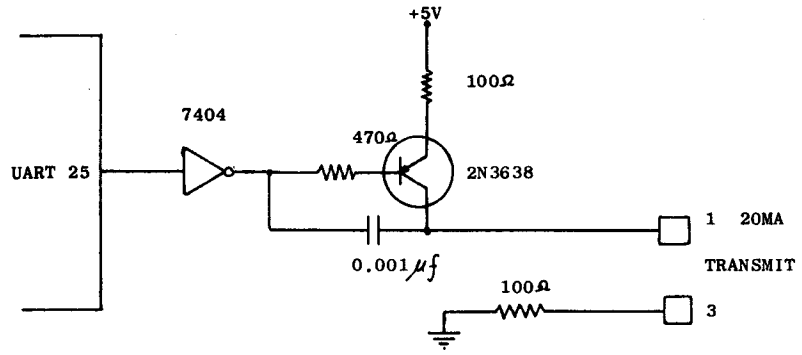
been addressed as a listener on the IEEE bus. Basic can now transfer the data one character at a time to the IEEE output routine which waits for the NRFD line to go low indicating that all the listening devices on the bus are ready to accept data. A single byte of data is then put onto the bus and a DAV pulse generated to indicate that valid data is now on the bus. The IEEE output routine then waits for the NDAC line to go high showing that the data has been received by the listener, however if the NDAC pulse is not received within 64 milliseconds of the NRFD line going low then an error is generated and bit 0 of the status byte is set indicating a listener time out. To stop the system returning to command mode immediately follow the PRINT command with a test for setting of bit 0 of the status byte. When all characters have been transferred by Basic the operating system transfers control to an IEEE end routine which sends an EOI pulse along with the last character stored in the output buffer in location 217 (old ROMs 246). Having done this an unlisten command is sent to the bus thereby freeing it for a subsequent operation and the default I/O device is reset to 3 thereby re-enabling output on the screen.

Having finished all inputs or outputs between a logical file in the PET and one or more devices on the IEEE bus, the file for each device must be closed. This is done by the CLOSE command, CLOSE 5 will close the device associated with logical file 5 by the OPEN command. On receipt of a CLOSE statement the operating system will send a listen command to the specified device followed by a secondary address command which is the OR of hex EO and the secondary address, signalling to the device that it should stop its current function and return to an initialised state.

It should be noted however that the operating system in old ROM PETs will not allow the LOAD and SAVE commands to be used with an IEEE device unless the program is transferred in an ASCII format. To SAVE a program onto an IEEE device with these machines one must list the program to that device. To get this ASCII program back again and perform the equivalent of a LOAD requires a technique identical to that used to merge two programs together, except that the default device number in location 175 should be set to the device number of the IEEE device rather than 1 which is the cassette device number. The LOAD and SAVE commands are available on all new ROM machines.

The commonest use of the IEEE bus is not to service instrumentation but simply to connect a printer to the PET. Unfortunately for this purpose the IEEE is not ideal since the majority of cheap printers use either an RS232 or a 20ma loop serial interface. The only way to overcome this is to construct an interface circuit which converts the parallel IEEE output to a serial output.

The circuit to do this is simple and can be constructed with very little expense. The circuit shown in Figure 5.6 performs three functions. Firstly it converts the PET ASCII code into stndard ASCII and generates the required control signals for the IEEE. Secondly a UART is used to convert the parallel data into serial data rate timing being provided by the 555 timer, this can be adjusted to the correct baud rate by the 50K potentiometer. Thirdly the serial output is converted to the correct levels in this case to those required for an RS232 interface, to give a 20ma loop interface then the following circuit is connected to pin 25 of the UART in place of the 75150 IC.



The interface circuit requires a separate power supply capable of providing the following voltages: +12 , +5 , -12 and ground or 0 volts, the current consumption of any of these voltages is low and a mains adaptor for a calculator could be used to provide the larger voltages, with the lower voltages derived from them. This circuit is only designed as an unidirectional interface, it is very much harder to construct a bi-directional interface owing to the strict timing requirements of the IEEE bus. Both upper and lower case characters are printed and the interface can be used to both list programs and print data, the command sequence to list a program is : OPEN 4,4 : CMD 4 : LIST and to print data : OPEN 4,4,4 : PRINT # 4, A or A$.

An application finding increasing use amongst scientists and engineers is to use the PET as a data logger by connecting one or more instruments to the IEEE bus and using the PET to sample, store and process data from these instruments. The problem encountered by most people when using the PET for this kind of application is the limitation on sampling speed imposed by Basic. In most applications this is crucial since the majority of physical events are fairly rapid lasting no more than a few seconds and the fewer the measurements over that period the more likely we are to lose vital information. For these reasons it is preferable to use machine code subroutines to transfer data from the measuring

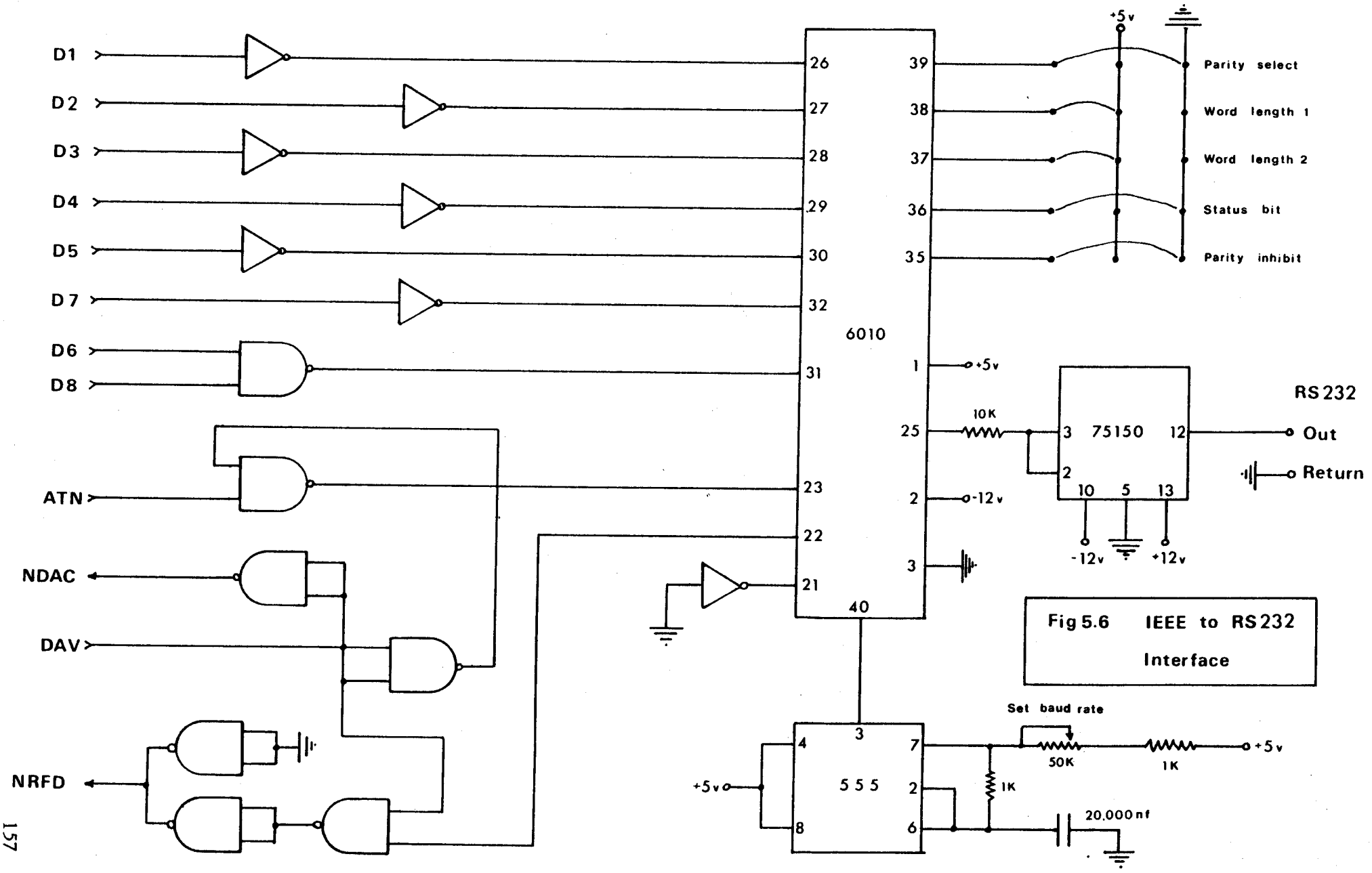


Fig 5.6 IEEE to RS232 Interface

instruments to the PET since these will allow data transfer rates in excess of 5000 bytes per second. At these transfer rates the problem is not sampling rates but data storage, since even with a 32K PET one can only store a few seconds of data at 5000 bytes per second, this can only be overcome by making a compromise between sampling rate and sampling period. Another way of reducing the quantity of data stored is to preprocess it as the data is entered and store only the required information or to sample short blocks of data and store each block on tape or disk.

The following programs are a set of machine code subroutines to handle data transfer between an IEEE instrument and the PET and they can be used as the basis for a wide variety of different data logging application programs.In these examples the programs are located from 6144 upwards with the top of the Basic memory area having been set at 6144 giving 5K for Basic programs and the top 2K for the machine code subroutines and data storage. The subroutines are configured to read data from just one device and store the data in memory from address 6401 upwards. The same routines can be used to obtain data from more than one device by changing the device numbers and alternating access between devices. The program controls a given number of data transfers between an IEEE device and the PET, each transfer consisting of one or more bytes - in this example eight bytes - the number of bytes can be changed by POKEing the required number into location 6200. Each data transfer is preceded by a GET - Group Execute Trigger - command on the IEEE bus and the IEEE device must be correctly addressed as a "talker" or a "listener" at all times by sending the correct MTA (My Talk Address) or MLA (My Listen Address) prior to the appropriate transfer.

Prior to loading the program the top of memory pointers must be lowered to prevent Basic overwriting the IEEE program and data, this is done at the beginning of the Basic program using the commands POKE 134,255 and POKE 135,23. The number of data transfers can be controlled by the contents of location 6400 which should be POKEd with the required value. The data obtained by this program is stored in locations 6401 upwards and can be retrieved by PEEKing from the Basic control program. The subroutines are limited to transferring only 256 bytes of data since the index registers are used for counting. The IEEE bus handshaking program can be called with the Basic command SYS(6144).

```
IEEE bus handshaking routine - main program
1800    A2 00       LDX # 00    prepare index register
1802    A9 FB       LDA # FB    set ATN low
1804    2D 40 E8    AND E840
1807    8D 40 E8    STA E840
180A    A9 28       LDA # 28    MLA (28 for this device)
```

```
180C    85 01       STA 01
180E    20 80 18    JSR 1880    handshake into bus
1811    A9 08       LDA # 08    GET
1813    85 01       STA 01
1815    20 80 18    JSR 1880    handshake
1818    A9 48       LDA # 48    MTA
181A    85 01       STA 01
181C    20 80 18    JSR 1880    handshake
181F    A9 FD       LDA # FD    set NRFD low
1821    2D 40 E8    AND E840
1824    8D 40 E8    STA E840
1827    A9 F7       LDA # F7    and NDAC low also
1829    2D 21 E8    AND E821
182C    8D 21 E8    STA E821
182F    A9 04       LDA # 04    set ATN high
1831    OD 40 E8    ORA E840
1834    8D 40 E8    STA E840
1837    AO 08       LDY # 08    ready to count 8 bytes
1839    20 BO 18    JSR 18BO    handshake data from bus
183C    A5 02       LDA 02      result to A
183E    9D 01 19    STA 1901,X  store in 1901+X
1841    E8          INX
1842    88          DEY
1843    DO F4       BNE 1839    jump if Y not zero
1845    A9 FB       LDA # FB    set ATN low
1847    2D 40 E8    AND E840
184A    8D 40 E8    STA E840
184D    A9 02       LDA # 02    set NRFD high
184F    OD 40 E8    ORA E840
1852    8D 40 E8    STA E840
1855    A9 08       LDA # 08    set NDAC high
1857    OD 21 E8    ORA E821
185A    8D 21 E8    STA E821
185D    A9 5F       LDA # 5F    UNT
185F    85 01       STA 01
1861    20 80 18    JSR 1880    handshake to bus
1864    A9 04       LDA # 04    set ATN high
1866    OD 40 E8    ORA E840
1869    8D 40 E8    STA E840
186C    CE 00 19    DEC 1900    decrease counter
186F    DO 91       BNE 1802    jump if not zero
1871    60          RTS         return to BASIC program
subroutine to handle handshake into bus
1880    AD 40 E8    LDA E840    NRFD?
1883    29 40       AND # 40
1885    FO F9       BEQ 1880    jump back if not ready
1887    A5 01       LDA 01      ready: get data byte
1889    49 FF       EOR # FF    complement it
188B    8D 22 E8    STA E822    send to bus
188E    A9 F7       LDA # F7    set DAV low
1890    2D 23 E8    AND E823
1893    8D 23 E8    STA E823
1896    AD 40 E8    LDA E840    NDAC?
1899    29 01       AND # 01
```

```
189B    F0 F9       BEQ 1896    jump back if not accepted
189D    A9 08       LDA # 08    accepted; set DAV high
189F    0D 23 E8    ORA E824
18A2    8D23 E8     STA E823
18A5    A9 FF       LDA # FF    255     into bus
18A7    8D 22 E8    STA E822
18AA    60          RTS         return to main

subroutine to handle handshake from bus
18B0    A9 02       LDA # 02    set NRFD high
18B2    0D 40 E8    ORA E840
1885    8D 40 E8    STA E840
18B8    AD 40 E8    LDA E840    DAV?
18BB    29 80       AND # 80
18BD    D0 F9       BNE 18B8    jump back if not valid
18BF    AD 20 E8    LDA E820    get data byte from bus
18C2    49 FF       EOR # FF    complement

18C4    85 02       STA 02      store in $0002
18C6    A9 FD       LDA # FD    set NRFD low
18C8    2D 40 E8    AND E840
18CB    8D 40 E8    STA E840
18CE    A9 08       LDA # 08    set NDAC high
18D0    0D 21 E8    ORA E821
18D3    8D 21 E8    STA E821
18D6    AD 40 E8    LDA E840    DAV high ?

18D9    29 80       AND # 80
18DB    F0 F9       BEQ 18D6    jump back if not
18DD    A9 F7       LDA # F7    set NDAC low
18DF    2D 21 E8    AND E821
18E2    8D 21 E8    STA E821
18E5    A9 FF       LDA # FF    255     into bus
18E7    8D 22 E8    STA E822
18EA    60          RTS         return to main
```

The Video Display

One of the virtues of the PET video display is the flexibility imparted to it by being a memory mapped design with the majority of the control being performed by software. This allows the user to manipulate the display in ways which would be impossible with a conventional terminal, as an example: most users will have used the POKE command in locations between 32768 and 32767 to move characters around the display.

To understand how the display can be used to produce certain effects, we must look at how the display is generated. There are two processes involved in generating the display, the first performed almost completely by hardware is the character generation and screen refresh which also involves the timing of the horizontal and vertical scan outputs to the video

monitor. The second process is done completely by software and involves taking a character from the keyboard buffer or from Basic output buffer and placing that character in a specific location in the display memory area. Though the character and raster generation is mostly performed by hardware, the user can besides writing characters into the screen memory directly control two (only one in dynamic PETs) of its functions via a couple of lines connected to the I/0 ports. The first is familiar to all users and is the conversion of the graphics mode into the lower case mode by toggling the CA2 line on the 6522 thus POKE 59468,14 will set the display in lower case mode and POKE 59468,12 will set it in graphics mode. The second function, only available in static RAM machines is provided by the CA2 line on 6520 #1 it blanks the screen during character entry and retrace. This prevents the broken characters which appear on the screen during PEEK and POKE operations, due to interferrence between the character generator addressing and the processor addressing. The screen blanking function can be used in machine code programs to give a nice clean display free of interference, in Basic programs it can be used to suppress the display until a whole screen full of information is present. The commands in Basic to produce screen blanking are POKE 59409,52 and to restore the display POKE 59409,60. There is a third very important connection between the character generation, display hardware and the processor, it is the 60Hz retrace input and is connected to both the CB1 interrupt input on 6520 #1 and I/0 line 5 on port B of the 6522. The function of this line is to generate a system interrupt which calls the routines for scanning the keyboard, updating the display and the clock TI. Called the retrace input because it is produced each time the raster scan reaches the bottom of the screen prior to the scan flyback, it is during flyback period that the screen is blanked and the display updated. The retrace interrupt can be disabled allowing the programmer to disable the keyboard also code can be inserted into the interrupt routines, these procedures have been dealt with in other sections of this book. The retrace input can also be used to perform the same function as the blank command in a machine code program, namely to suppress screen interference while writing to screen memory. This can be done by waiting for the retrace input to the 6522 with the interrupts disabled. A simple subroutine to do this would be as follows:

```
AD E8 40    RETRACE   LDA E840      :pu  port B in accumulator
49 20                 EOR #$20      :mask off line 5
29 20                 AND #$20      :is line 5 set
F0 F7                 BEQ RETRACE   :if not return to RETRACE
60                    RTS           :return to calling program
```

Data is displayed on the screen by the operating system either as a result of an entry on the keyboard or the execution of a print statement in a Basic program. Data can also come from other sources like input prompts, error messages or from a machine code program or POKE command writing directly to the display memory. When a key is pressed the operating system translates the matrix co-ordinates into an ASCII character which is stored in a "first in first out" queue located in the 10 byte keyboard buffer located at 623 to 632. The operating system then periodically empties this buffer byte by byte into the display memory at a location pointed to by the current position of the cursor. The ASCII code is also converted at this time to the slightly different version used by the character generator obtained by dropping bit 6. The position of the cursor is stored as two values. The first is the cursor column position which is stored in location 198. The second is the pointer to the start of the line of the cursor location this is stored as a two byte number in location 196 and 197, the value stored being the number of characters from the beginning of the screen. The blinking cursor is controlled as part of the retrace interrupt subroutine and is activated when the PET is either in the command mode or the input mode, its activation being controlled by the contents of location 170. When the cursor is active the contents of 170 is 0 and when inactive 1, by executing a POKE 170,0 prior to a GET command we can use this location to give us a blinking cursor for the GET input a feature normally absent.

The display itself is only 40 characters wide, a limitation which is dictated by the size of the screen and the display circuitry bandwidth. Since many applications require a line length longer than 40 characters, the PET operating system allows lines up to 80 characters to be displayed by folding the display back onto the next line. To allow lines of up to 80 characters and yet avoid leaving empty lines where the previous line has less than 40 characters the operating system uses a table of pointers to the beginning of each line. Each line has a pointer which indicates whether it is the beginning of a new line or the continuation of the previous line. These pointers are stored in a table in memory locations 224 - 248 there being one entry for each of the 25 lines on the screen. The contents of these pointers is the least signficant byte of the start address on the screen with the status of bit 7 indicating whether it is a new or continuation line. Thus whenever the cursor is moved up or down the operating system will examine the status of the line on which the cursor currently lies and initialise the PET to the proper line number so that when a carriage return is pressed the cursor will jump down the appropriate

number of lines.

The fact that the PET has only a 40 character by 25 line display can be rather limiting when trying to display a graph with a reasonable resolution between plotted points. A technique can be used which doubles the density of the display to give an 80 by 50 dot picture. This is done using the quarter character graphics in place of the normal full character graphics. One of a range of full sized graphics characters are used in each character space to simulate a 2 x 2 matrix, the full sized character used depends on the contents of that matrix. A program to plot a display using quarter characters must be able to plot by its x and y co-ordinates and also selectively erase characters and replace them with new characters when the plot within a single character matrix is changed. Such a program written in Basic would take a considerable time to construct a display and it is much quicker and easier to write it in machine code, which is then called from a Basic program as and when needed. The following program which is stored in the second cassette buffer performs such a function - the first listing is an assembled version of the machine code program, the second is the same program with a Basic loader and a subroutine to draw lines on the screen.

The program which will work on both old and new ROM machines can be called with the command SYS(826) from Basic with the X co-ordinate stored using POKE 48,X and the Y co-ordinate using POKE 49,Y. The Basic machine code loader used in this program lies between lines 200 and 400 and is a very useful means of entering machine code into the PET using the standard hexadecimal values. The data required by the screen drawing subroutine are the X and Y co-ordinates of the starting and ending points of the line with the 0,0 co-ordinates being in the bottom right corner, and the 49,79 co-ordinate in the top left.

```
                    :PROGRAM TO PLOT POINTS ON
                    :PET IN DOUBLE DENSITY FORMAT
                    :X-COORD IN LOCATION 48 (30)
                    :Y-COORD IN LOCATION 49 (31 &32)
                    :0 IN LOCATION 51 (33) TO ADD
                    :1 IN LOCATION 51 (33) DELETE
                    :ERROR FLAG IN LOCATION 998 (3E6)
                    :1 OR 2 PLOT OUT OF RANGE
                    :4 NON-PLOTTABLE CHARACTER ALREADY
                    :AT THESE COORDINATES ON SCREEN
                    :
                    REFRESH = E840
                    XCOORD  = 30
                    YCOORD  = 31
                    AORD    = 33
                    BINOFF  = 34
                    :
0000                     *  = 033A
```

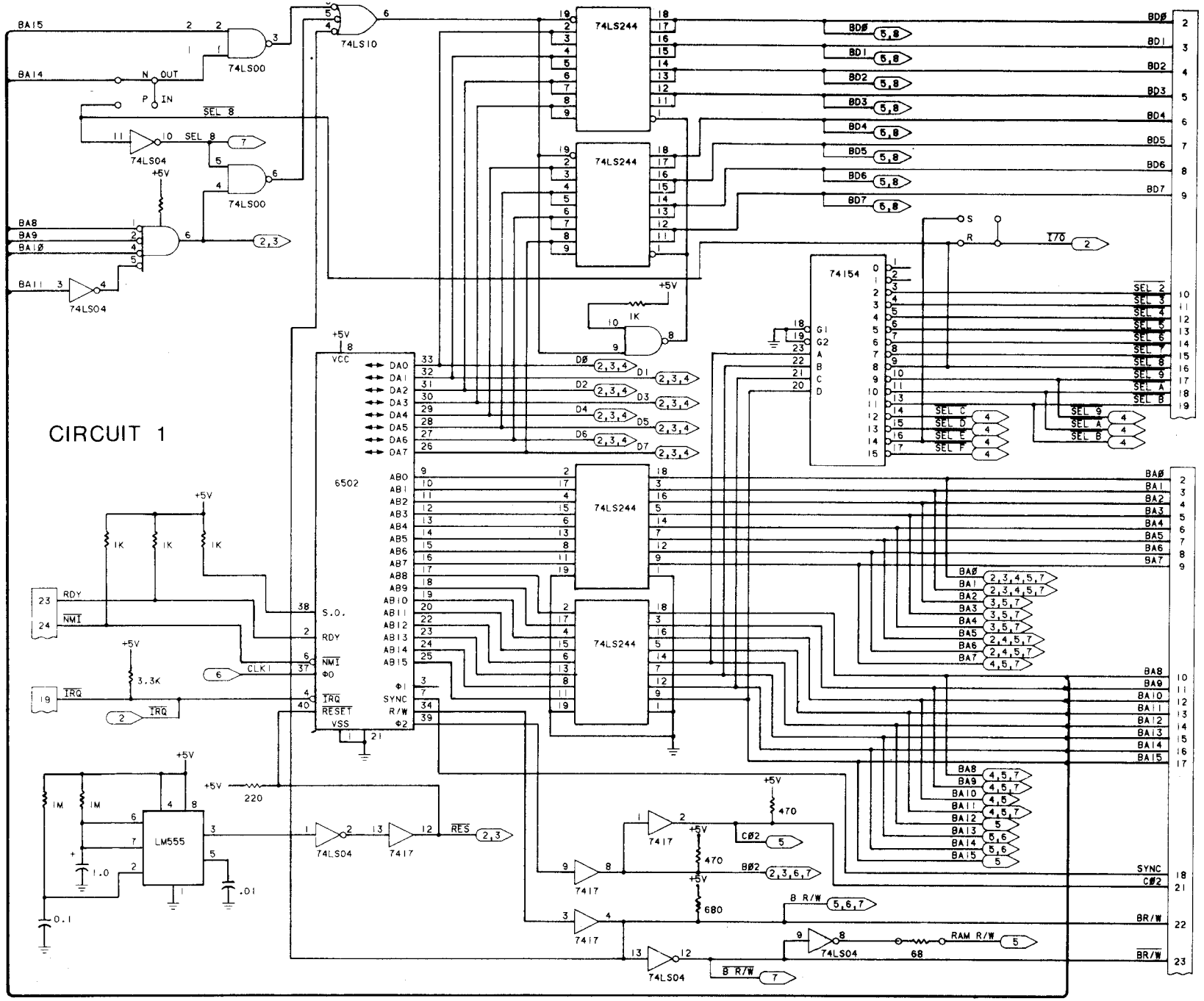
```
033A  A9  00        START   LDA $0
033C  8D  E6  03            STA ERROR
033F  85  34                STA BINOFF
                    :TEST IF YCOORD  49
0341  A5  31                LDA YCOORD
0343  C9  32                CMP $50
0345  90  03                BCC YOK
0347  EE  E6  03            INC ERROR
                    :TEST IF XCOORD  79
034A  A5  30        YOK     LDA XCOORD
034C  C9  50                CMP $80
034E  90  03                BCC XOK
0350  EE  E6                INC ERROR
                    :RETURN IG OUT-OF-RANGE ERROR
0353  2C  E6  03    XOK     BIT ERROR
0356  F0  01                BEQ SORIG
0358  60                   RTS
                    :INVERT SCREEN FROM TOP TO BOTTOM
                    :(Y-COORDINATE)
0359  A9  31        SORIG   LDA $49
035B  38                    SEC
0035C E5  31                SBC YCOORD
035E  85  31                STA YCOORD
                    :SAVE BOTTOM BIT OF X COORD IN
                    :BINOFF
0363  46  30                LSR XCOORD
0362  26  34                ROL BINOFF
                    :SAVE BOTTOM BIT OF Y COORD IN
                    :BINOFF
0364  46  31                LSR YCOORD
0366  26  34                ROL BINOFF
                    :MULTIPLY YCOORD BY 40 AND
                    :ADD SCREEN BASE ADDRESS
0368  06  31                ASL YCOORD
036A  06  31                ASL YCOORD
036C  06  31                ASL YCOORD
036E  A5  31                LDA YCOORD :SAVE IN A-REG
0373  06  31                ASL YCOORD
0372  26  32                ROL YCOORD+1
0374  06  31                ASL YCOORD
0376  26  32                ROL YCOORD+1
0378  18                    CLC
0379  65  31                ADC YCOORD
037B  85  31                STA YCOORD
037D  A5  32                LDA YCOORD+1
037F  69  C0                ADC $ C0 :START OF SCREEN
0381  85  32                STA YCOORD+1
                    :EXPAND BINOFF
0383  A6  34                LDX BINOFF
0385  A9  01                LDA $1
0387  85  34                STA BINOFF
0389  E0  00        EXP     CPX $0
038B  F0  05                BEQ ENDEXP
038D  06  34                ASL BINOFF
```

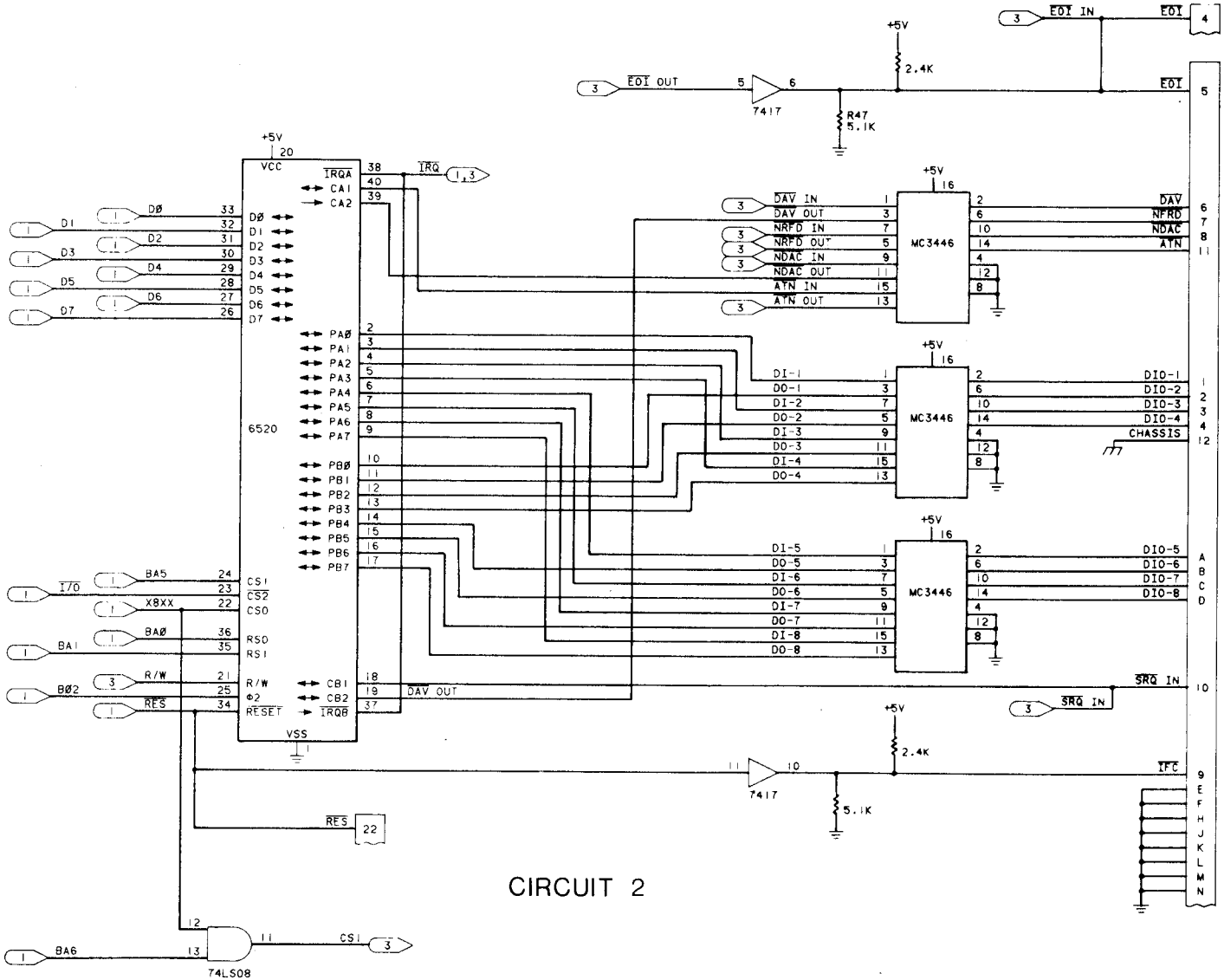
```
038F  CA                        DEX
0393  90   F7                 BCC EXP
                          :GET CHAR INTO A-REG
0392  A4   30           ENDEXP LDY XCOORD
0394  B1   31                 LDA (YCOORD),Y
                          :CHECK CHAR IS VALID GRAPHIC
0396  A2   00                 LDX $0
0398  DD   CE   03      MOREC  CMP TABLE.X
039B  FO   OB                 BEQ FOUND
039D  E8                      INX
039E  EO   10                 CPX $16
03AO  90   F6                 BCC MOREC
03AO  90   F6                 BCC MOREC
                          :CHAR IS NOT IN TABLE
03A2  A9  .04                 LDA $4
03A4  8D   E6   03            STA ERROR
03A7  60                      RTS
                          :CHAR IS IN TABLE
                          :SO ERASE OR ADD
03A8  A5   33           FOUND  LDA AORD
03AA  DO   07                 BNE ERASPT
                          :ADD POINT TO SCREEN
03AC  8A                ADDPT  TXA
03AD  05   34                 ORA BINOFF
03AF  18                      CLC
03BO  AA                      TAX
03B1  90   OA                 BCC WAIT
                          :ERASE POINT FROM SCREEN
03B3  A5   34           ERASPT LDA BINOFF
03B5  49   FF                 EOR $ FF
03B7  85   34                 STA BINOFF
03B9  8A                      TXA
03BA  25   34                 AND BINOFF
03BC  AA                      TAX
                          :WAIT FOR SCREEN REFRESH
03BD  AD   40   E8      WAIT   LDA REFRESH
03CO  49   20                 EOR $ 20
03C2  29   20                 AND $ 20
03C4  FO   F7                 BEQ WAIT
                          :WRITE NEW CHAPHIC CHAR TO SCREEN
03C6  BD   CE   03            LDA TABLE.X
03C9  A4   30                 LDY XCOORD
03CB  91   31                 STA (YCOORD).Y
                          :RETURN SUCCESSFULLY
03CD  60                      RTS
                          :TABLE OF GRAPHICS CHARACTERS
03CE        20          TABLE  . BYTE 20. 7E. 7B. 61
03CF        7E
03DO        7B
03D1        61
03D2        7C          .BYTE 7C. E2. FF. EC.
03D3        E2
03D4        FF
03D5        EC
```

```
 03D6        6C           .BYTE 6C. 7F. 62. FC
 03D7        7F
03D8         62
 03D9        FC
 03DA        E1           .BYTE E1. FB. FE. AO
 03DB        FB
 03DC        FE
 03DD        AO
 03DE                     *=*+8
 03E6             ERROR   *=*+1
 03E7
```

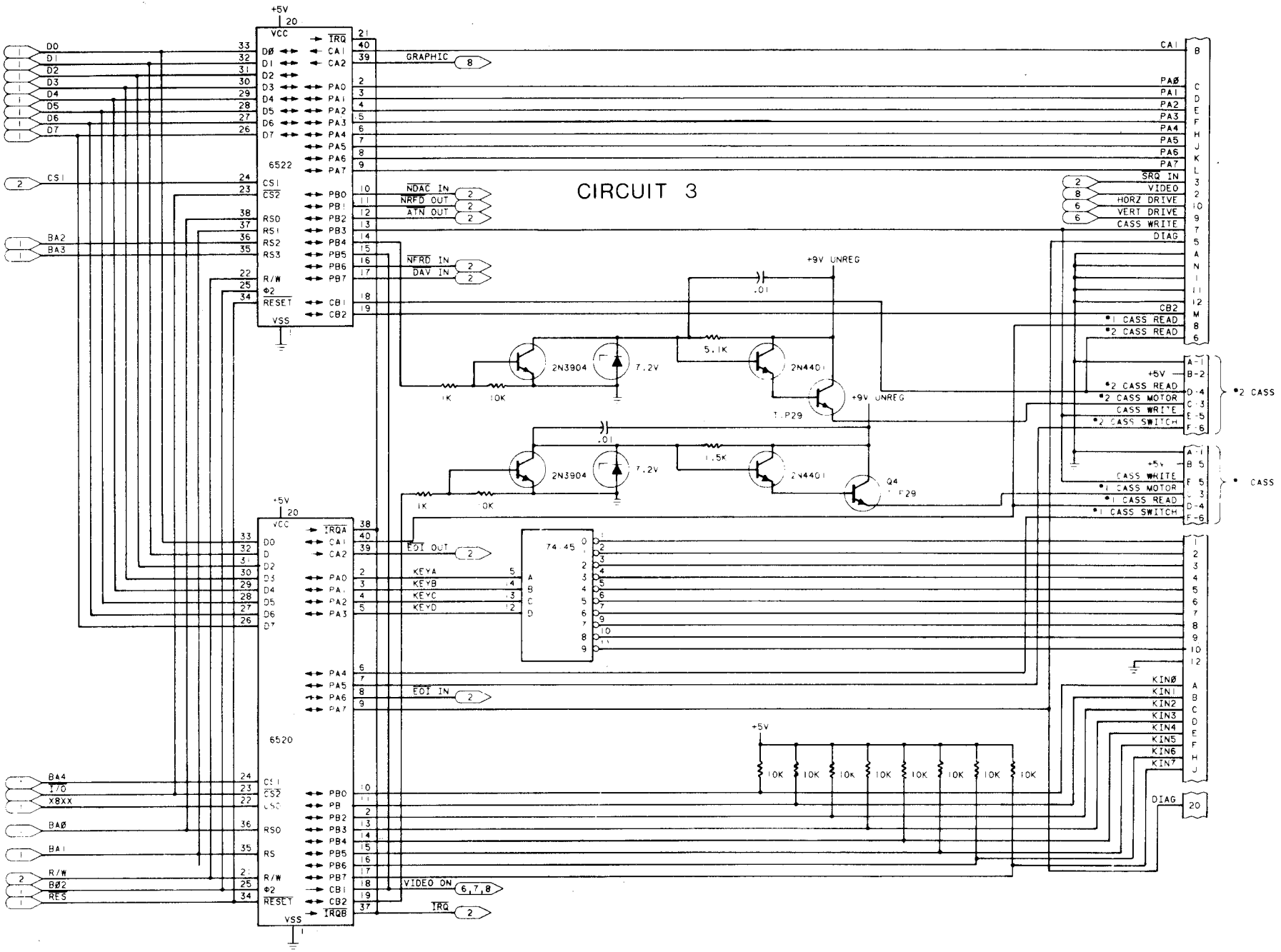
```
10 DATA826
20 DATA A9,00,8D,E6,03,85,34
30 DATA A5,31,C9,32,90,03,EE,E6,03
40 DATA A5,30,C9,50,90,03,EE,E6,03
50 DATA 2C,E6,03,F0,01,60
60 DATA A9,31,38,E5,31,85,31
70 DATA 46,30,26,34
80 DATA 46,31,26,34
90 DATA 06,31,06,31,06,31,A5,31,06,31,26,32,06,31,26,32,18,65
91 DATA 31,85,31,A5,32,69,80,85,32
100 DATA A6,34,A9,01,85,34,E0,00,F0,05,06,34,CA,90,F7
110 DATA A4,30,B1,31,A2,00,DD,CE,03,F0,0B,E8,E0,10,90,F6
120 DATA A9,04,8D,E6,03,60
130 DATA A5,33,D0,07,8A,05,34,18,AA,90,0A
140 DATA A5,34,49,FF,85,34,8A,25,34,AA
150 DATA AD,40,E8,49,20,29,20,F0,F7
160 DATA BD,CE,03,A4,30,91,31,60
170 DATA 20,7E,7B,61,7C,E2,FF,EC,6C,7F,62,FC,E1,FB,FE,A0
180 DATA*
200 READ L
210 READ A$
220 C=LEN(A$)
230 IF A$="*" THEN 400
240 IF C<1 OR C>2 THEN330
250 A=ASC(A$)-48
260 B=ASC(RIGHT$(A$,1))-48
270 N=B+7*(B>9)-(C=2)*(16*(A+7*(A>9)))
280 IF N<0 OR N>255 THEN 320
290 POKE L,N
300 L=L+1
310 GOTO210
320 PRINT"BYTE"L"= "A$"  ???"
400 PRINT" ";
1000 PRINT"";
1005 INPUTX1,Y1,X2,Y2
1010 GOSUB2000
1015 PRINT"                              "
1020 GOTO1000
```

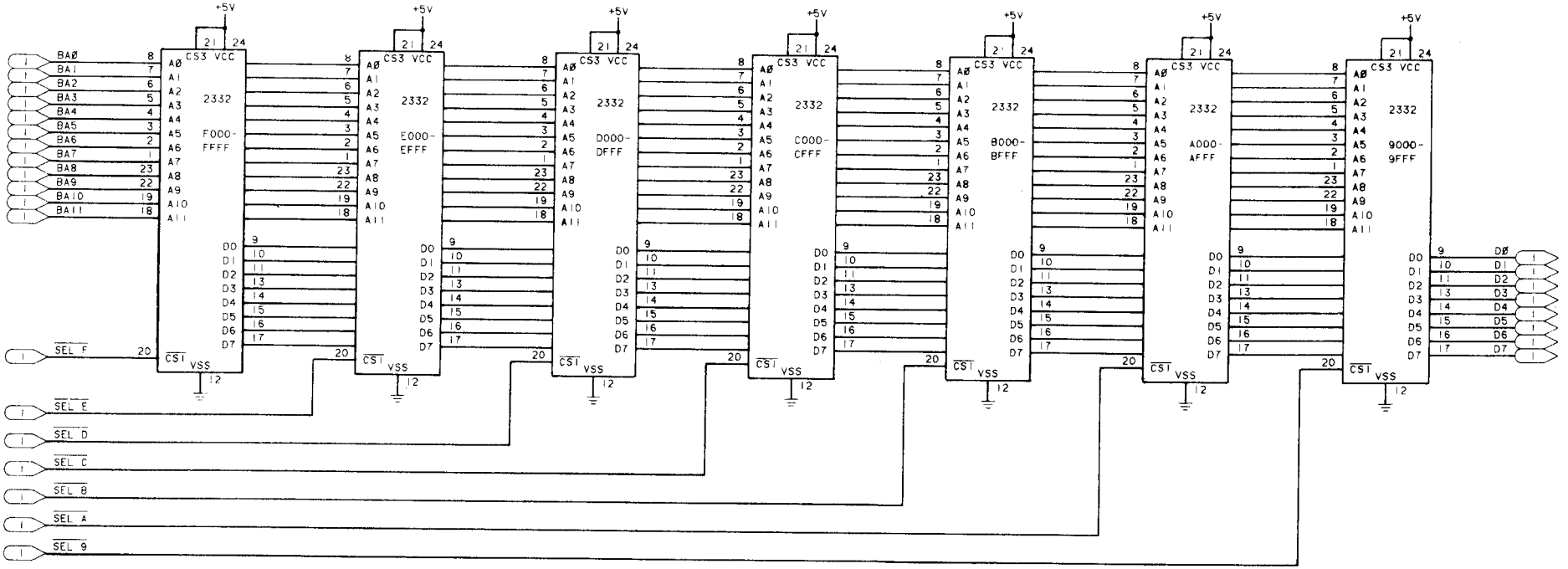
```
2000 REM SUBROUTINE TO DRAW LINE
2010 REM BETWEEN TWO POINTS ON SCREEN
2020 REM CHECK COORDINATES IN BOUNDS
2030 IF(X1>=0ANDX1<=79)AND(X2>=0ANDX2<=79)THEN 2060
2040 ER$="X OUT OF RANGE"
2050 RETURN
2060 IF(Y1>=0ANDY1<=49)AND(Y2>=0ANDY2<=49)THEN 2090
2070 ER$="Y OUT OF RANGE"
2080 RETURN
2090 ER$=""
2100 XD=X2-X1
2110 YD=Y2-Y1
2120 REM NEAREST DIAGONAL
2130 A0=1:A1=1
2140 IFYD<0THENA0=-1
2150 IFXD<0THENA1=-1
2160 REM NEAREST HORIZ/VERT
2170 XE=ABS(XD):YE=ABS(YD):D1=XE-YE
2180 IFD1>=0THEN2220
2190 S0=-1:S1=0:LG=YE:SH=XE
2200 IFYD>=0THENS0=1
2210 GOTO2240
2220 S0=0:S1=-1:LG=XE:SH=YE
2230 IFXD>=0THENS1=1
2240 REM SET UP
2250 TT=LG:TS=SH:UD=LG-SH:CT=SH-LG/2
2255 D=0
2260 REM WHILE MORE POINTS DO
2270 POKE48,X1:POKE49,Y1:POKE50,0:POKE51,D:SYS(826)
2280 IFCT>=0THEN2320
2290 CT=CT+TS:X1=X1+S1:Y1=Y1+S0
2310 GOTO2360
2320 CT=CT-UD:X1=X1+A1:Y1=Y1+A0
2360 TT=TT-1
2370 IF TT<0THEN RETURN
2380 GOTO2270
```


CIRCUIT 1
74LS10
74LS00
74LS04
74LS244
74154
6502
LM555
7417
BA15
BA14
SEL 8
BA8
BA9
BA10
BA11
BD0
BD1
BD2
BD3
BD4
BD5
BD6
BD7
SEL 2
SEL 3
SEL 4
SEL 5
SEL 6
SEL 7
SEL 8
SEL 9
SEL A
SEL B
SEL C
SEL D
SEL E
SEL F
I/O
S
R
+5V
1K
3.3K
220
470
680
68
1M
1.0
.01
0.1
RDY
NMI
IRQ
RESET
CLK1
S.O.
SYNC
R/W
VCC
VSS
RAM R/W
B R/W
BR/W
CØ2
BØ2
SYNC

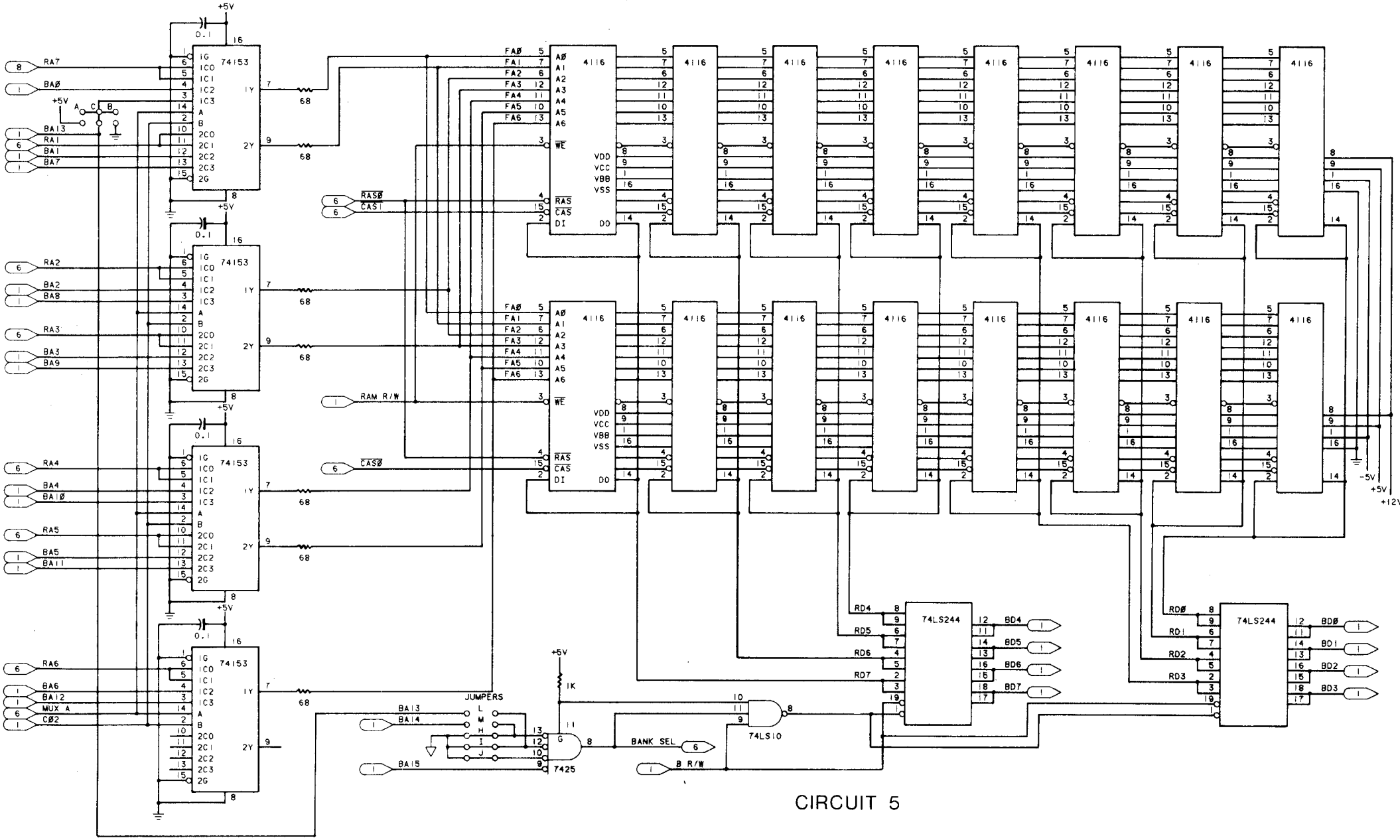

6520
MC3446
7417
74LS08
+5V
2.4K
5.1K
R47
5.1K
VCC
VSS
EOI IN
EOI OUT
DAV IN
DAV OUT
NRFD IN
NRFD OUT
NDAC IN
NDAC OUT
ATN IN
ATN OUT
DAV
NRFD
NDAC
ATN
DIO-1
DIO-2
DIO-3
DIO-4
DIO-5
DIO-6
DIO-7
DIO-8
CHASSIS
SRQ IN
IFC
IRQ
RES
R/W
Φ2
BA0
BA1
BA5
BA6
BØ2
I/O
X8XX
CS1
CS2
CS0
RS0
RS1
IRQA
IRQB
CA1
CA2
CB1
CB2
RESET
DAV OUT

CIRCUIT 2


CIRCUIT 3
6522
6520
74 45
2N3904
2N4401
7.2V
5.1K
1.5K
1K
10K
.01
+5V
+9V UNREG
VCC
VSS
RESET
R/W
CS1
CS2
IRQ
IRQA
IRQB
GRAPHIC
NDAC IN
NRFD OUT
ATN OUT
NRFD IN
DAV IN
EOI OUT
EOI IN
KEYA
KEYB
KEYC
KEYD
VIDEO ON
SRQ IN
VIDEO
HORZ DRIVE
VERT DRIVE
CASS WRITE
DIAG
CASS READ
CASS MOTOR
CASS SWITCH
KIN0
KIN1
KIN2
KIN3
KIN4
KIN5
KIN6
KIN7
Q4
T-P29

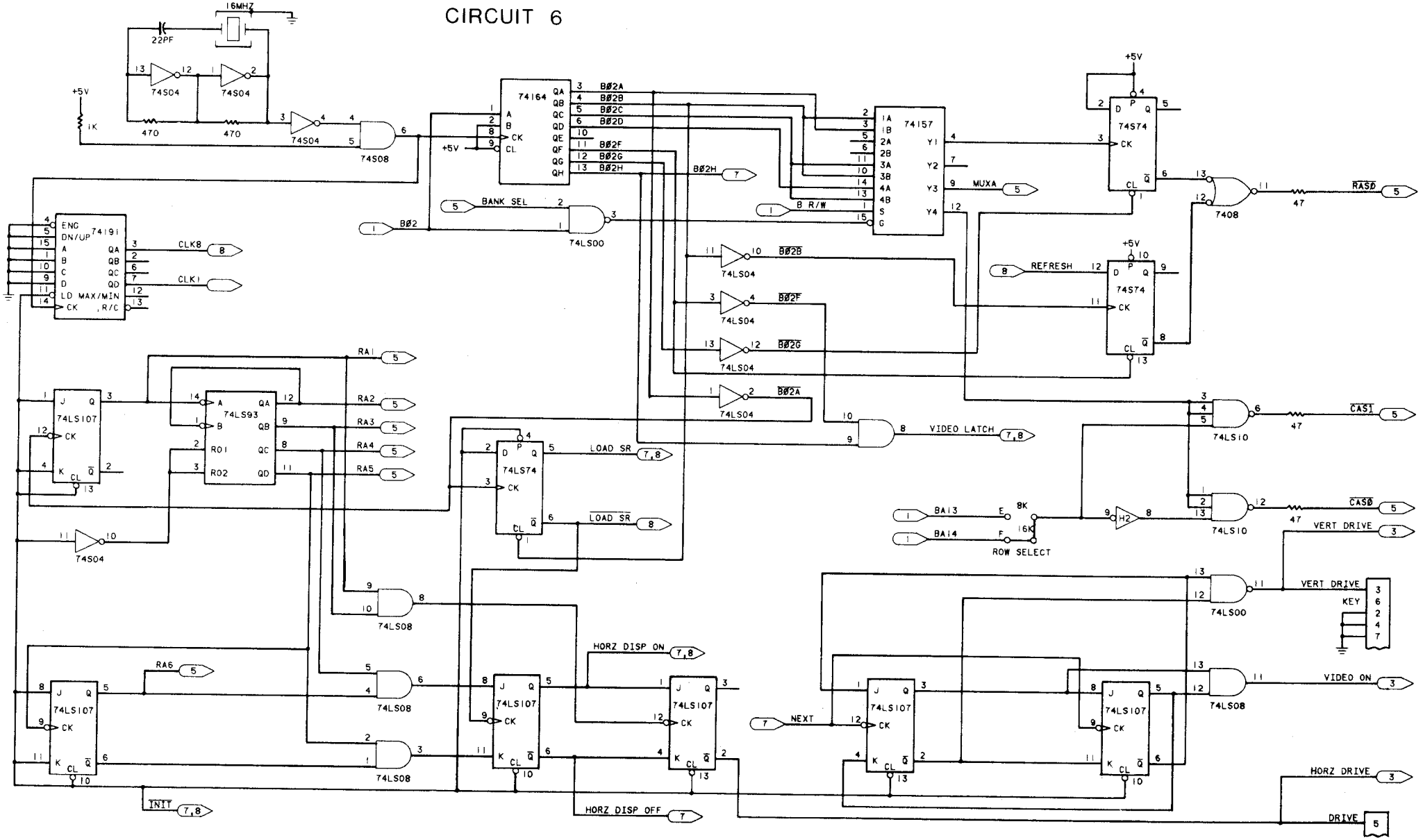

+5V
BA0
BA1
BA2
BA3
BA4
BA5
BA6
BA7
BA8
BA9
BA10
BA11
CS3 VCC
2332
F000-FFFF
E000-EFFF
D000-DFFF
C000-CFFF
B000-BFFF
A000-AFFF
9000-9FFF
D0
D1
D2
D3
D4
D5
D6
D7
CS1 VSS
SEL F
SEL E
SEL D
SEL C
SEL B
SEL A
SEL 9

CIRCUIT 4

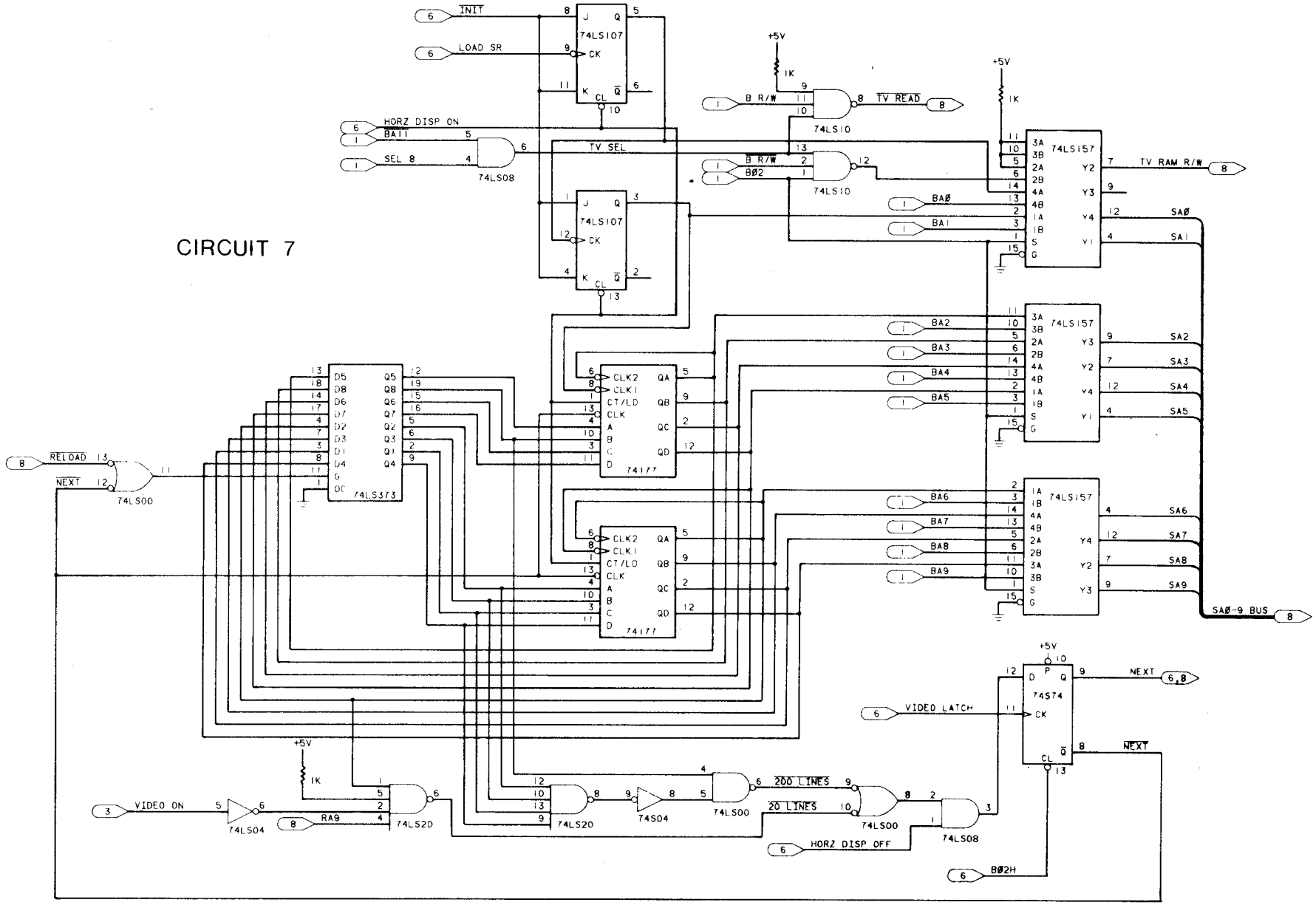

+5V
0.1
74153
1G
2G
1C0
1C1
1C2
1C3
A
B
2C0
2C1
2C2
2C3
1Y
2Y
RA7
BAØ
BA13
RA1
BA1
BA7
RA2
BA2
BA8
RA3
BA3
BA9
RA4
BA4
BA1Ø
RA5
BA5
BA11
RA6
BA6
BA12
MUX A
CØ2
68
RASØ
CAS1
RAM R/W
CASØ
FAØ
FA1
FA2
FA3
FA4
FA5
FA6
4116
AØ
A1
A2
A3
A4
A5
A6
WE
RAS
CAS
DI
DO
VDD
VCC
VBB
VSS
-5V
+12V
RD4
RD5
RD6
RD7
74LS244
BD4
BD5
BD6
BD7
RDØ
RD1
RD2
RD3
BDØ
BD1
BD2
BD3
1K
JUMPERS
L
M
H
I
J
BA14
BA15
7425
74LS10
BANK SEL
B R/W

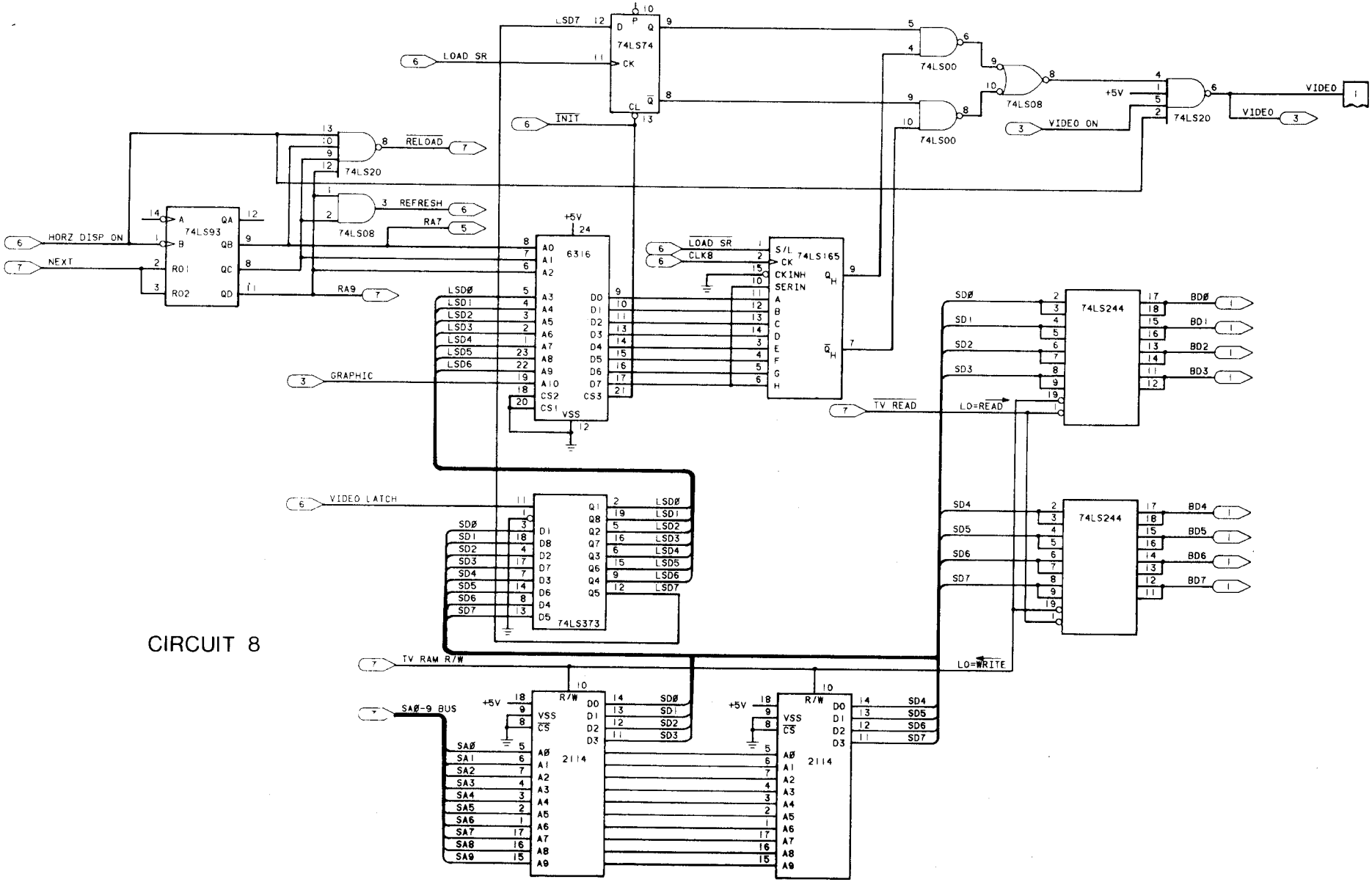
CIRCUIT 5

CIRCUIT 6


16MHZ
22PF
74S04
74S04
74S04
74S08
1K
470
470
+5V
74164
BØ2A
BØ2B
BØ2C
BØ2D
BØ2F
BØ2G
BØ2H
74157
MUXA
B R/W
74S74
74S74
7408
RASØ
47
74191
ENG
DN/UP
LD MAX/MIN
CK
R/C
CLK8
CLK1
BANK SEL
BØ2
74LS00
74LS04
74LS04
74LS04
74LS04
REFRESH
74S74
74LS10
CASI
74LS10
CASØ
BA13
BA14
8K
16K
ROW SELECT
VERT DRIVE
KEY
3
6
2
4
7
74LS00
VERT DRIVE
VIDEO ON
74LS08
HORZ DRIVE
DRIVE
74LS107
74LS93
RA1
RA2
RA3
RA4
RA5
RA6
74LS08
74LS08
74LS08
74LS04
74LS107
74LS74
LOAD SR
LOAD SR
VIDEO LATCH
HORZ DISP ON
HORZ DISP OFF
INIT
NEXT
74LS107
74LS107
74LS107

CIRCUIT 7


INIT
LOAD SR
74LS107
HORZ DISP ON
BAII
SEL 8
74LS08
TV SEL
+5V
1K
B R/W
TV READ
74LS10
BØ2
74LS157
TV RAM R/W
BAØ
BA1
BA2
BA3
BA4
BA5
BA6
BA7
BA8
BA9
SAØ
SA1
SA2
SA3
SA4
SA5
SA6
SA7
SA8
SA9
SAØ-9 BUS
RELOAD
NEXT
74LS00
74LS373
74177
74S74
VIDEO LATCH
VIDEO ON
RA9
74LS04
74LS20
74S04
200 LINES
20 LINES
HORZ DISP OFF
BØ2H


74LS74
74LS00
74LS00
74LS08
74LS20
74LS20
74LS08
74LS93
74LS165
6316
74LS244
74LS244
74LS373
2114
2114
R/W
LOAD SR
INIT
RELOAD
REFRESH
RA7
RA9
HORZ DISP ON
NEXT
GRAPHIC
VIDEO LATCH
TV RAM R/W
SA0-9 BUS
LOAD SR
CLK8
TV READ
LO=READ
LO=WRITE
VIDEO
VIDEO ON
+5V
VSS
CS
LSD0
LSD1
LSD2
LSD3
LSD4
LSD5
LSD6
LSD7
SD0
SD1
SD2
SD3
SD4
SD5
SD6
SD7
BD0
BD1
BD2
BD3
BD4
BD5
BD6
BD7

CIRCUIT 8

6502 Instruction Set – Hex and Timing

		IMPLIED			ACCUM.			ABSOLUTE			ZERO PAGE			IMMEDIATE			ABS. X		
MNEMONIC		OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#
A D C	(I)							6D	4	3	65	3	2	69	2	2	7D	4	3
A N D	(I)							2D	4	3	25	3	2	29	2	2	3D	4	3
A S L					OA	2	I	OE	6	3	O6	5	2				IE	7	3
B C C	(2)																		
B C S	(2)																		
B E Q	(2)																		
B I T								2C	4	3	24	3	2						
B M I	(2)																		
B N E	(2)																		
B P L	(2)																		
B R K		OO	7	I															
B V C	(2)																		
B V S	(2)																		
C L C		I8	2	I															
C L D		D8	2	I															
C L I		58	2	I															
C L V		B8	2	I															
C M P								CD	4	3	C5	3	2	C9	2	2	DD	4	3
C P X								EC	4	3	E4	3	2	EO	2	2			
C P Y								CC	4	3	C4	3	2	CO	2	2			
D E C								CE	6	3	C6	5	2				DE	7	3
D E X		CA	2	I															
D E Y		88	2	I															
E O R	(I)							4D	4	3	45	3	2	49	2	2	5D	4	3
I N C								EE	6	3	E6	5	2				FE	7	3
I N X		E8	2	I															
I N Y		C8	2	I															
J M P								4C	3	3									
J S R								2O	6	3									
L D A	(I)							AD	4	3	A5	3	2	A9	2	2	BD	4	3
L D X	(I)							AE	4	3	A6	3	2	A2	2	2			
L D Y	(I)							AC	4	3	A4	3	2	AO	2	2	BC	4	3
L S R					4A	2	I	4E	6	3	46	5	2				5E	7	3
N O P		EA	2	I															
O R A								OD	4	3	O5	3	2	O9	2	2	ID	4	3
P H A		48	3	2															
P H P		O8	3	I															
P L A		68	4	I															
P L P		28	4	I															
R O L					2A	2	I	2E	6	3	26	5	2				3E	7	3
R O R					6A	2	I	6E	6	3	66	5	2				7E	7	3
R T I		4O	6	I															
R T S		6O	6	I															
S B C	(I)							ED	4	3	E5	3	2	E9	2	2	FD	4	3
S E C		38	2	I															
S E D		F8	2	I															
S E I		78	2	I															
S T A								8D	4	3	85	2					9D	5	3
S T X								8E	4	3	86	2							
S T Y								8C	4	3	84	2							
T A X		AA	2	I															
T A Y		A8	2	I															
T S X		BA	2	I															
T X A		8A	2	I															
T X S		9A	2	I															
T Y A		98	2	I															

ABS. Y			(IND. X)			(IND)Y			Z.PAGE,X			RELATIVE			INDIRECT			Z.PAGE,Y			PROCESSOR STATUS CODES
OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	OP	n	#	N V BDIZC
79	4	3	6I	6	2	7I	5	2	75	4	2										• • •• • • • ••
39	4	3	2I	6	2	3I	5	2	35	4	2										
									I6	6	2										
												9O	2	2							
												BO	2	2							
												FO	2	2							•
												3O	2	2							
												DO	2	2							
												IO	2	2							
												5O	2	2							
												7O	2	2							
D9	4	3	CI	6	2	DI	5	2	D5	4	2										• •• • •• • ••
									D6	6	2										• • • • • •
59	4	3	4I	6	2	5I	5	2	55	4	2										• • • •
									F6	6	2										
89	4	3	AI	6	2	8I	5	2	B5	4	2				6C	5	3				• • • •
BE	4	3							B4	4	2							B6	4	2	• • • • • •
									56	6	2										
I9	4	3	OI	6	2	II	5	2	I5	4	2										• •
									36	6	2										• • • • • • • • • • ••
F9	4	3	EI	6	2	FI	5	2	76	6	2										• •• • • • • • • • • • ••
									F5	4	2										
99	5	3	8I	6	2	9I	6	2	95	4	2							96	4	2	• •
									94	4	2										
																					• • • • • • • •

HEXADECIMAL CONVERSION TABLE

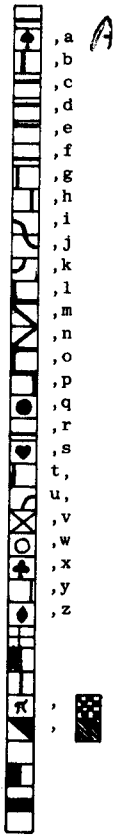
HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

5		4		3		2		1		0	
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0		0		0		0	0	0	0
1	1,048,576	1	65,536	1	4,096	1	256	1	16	1	1
2	2,097,152	2	131,072	2	8,192	2	512	2	32	2	2
3	3,145,728	3	196,608	3	12,288	3	768	3	48	3	3
4	4,194,304	4	262,144	4	16,384	4	1,024	4	64	4	4
5	5,242,880	5	327,680	5	20,480	5	1,280	5	80	5	5
6	6,291,456	6	393,216	6	24,576	6	1,536	6	96	6	6
7	7,340,032	7	458,752	7	28,672	7	1,792	7	112	7	7
8	8,388,608	8	524,288	8	32,768	8	2,048	8	128	8	8
9	9,437,184	9	589,824	9	36,864	9	2,304	9	144	9	9
A	10,485,760	A	655,360	A	40,960	A	2,560	A	160	A	10
B	11,534,336	B	720,896	B	45,056	B	2,816	B	176	B	11
C	12,582,912	C	786,432	C	49,152	C	3,072	C	192	C	12
D	13,631,488	D	851,968	D	53,248	D	3,328	D	208	D	13
E	14,680,064	E	917,504	E	57,344	E	3,584	E	224	E	14
F	15,728,640	F	983,040	F	61,440	E	3,840	F	240	F	15

Table of PET Codes

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL
0	00		@	end-line	BRK	0
1	01		A		ORA(I,X)	1
2	02		B			2
3	03		C			3
4	04		D			4
5	05		E		ORA Z	5
6	06		F		ASL Z	6
7	07		G			7
8	08		H		PHP	8
9	09		I		ORA #	9
10	0A		J		ASL A	10
11	0B		K			11
12	0C		L			12
13	0D	car ret	M		ORA	13
14	0E		N		ASL	14
15	0F		O			15
16	10		P		BPL	16
17	11	cur down	Q		ORA(I),Y	17
18	12	reverse	R			18
19	13	cur home	S			19
20	14	delete	T			20
21	15		U		ORA Z,X	21
22	16		V		ASL Z,X	22
23	17		W			23
24	18		X		CLC	24
25	19		Y		ORA Y	25
26	1A		Z			26
27	1B		[			27
28	1C		\			28
29	1D	cur right	]		ORA X	29
30	1E		↑		ASL X	30
31	1F		←			31
32	20	space	space	space	JSR	32
33	21	!	!	!	AND(I,X)	33
34	22	"	"	"		34
35	23	#	#	#		35
36	24	$	$	$	BIT Z	36
37	25	%	%	%	AND Z	37
38	26	&	&	&	ROL Z	38
39	27	'	'	'		39
40	28	(	(	(	PLP	40
41	29	)	)	)	AND #	41
42	2A	*	*	*	ROL A	42
43	2B	+	+	+		43
44	2C	,	,	,	BIT	44
45	2D	-	-	-	AND	45
46	2E	.	.	.	ROL	46
47	2F	/	/	/		47
48	30	Ø	Ø	Ø	BMI	48
49	31	1	1	1	AND(I),Y	49

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL
50	32	2	2	2		50
51	33	3	3	3		51
52	34	4	4	4		52
53	35	5	5	5	AND Z,X	53
54	36	6	6	6	ROL Z,X	54
55	37	7	7	7		55
56	38	8	8	8	SEC	56
57	39	9	9	9	AND Y	57
58	3A	:	:	:	CLI	58
59	3B	;	;	;		59
60	3C					60
61	3D	=	=	=	AND X	61
62	3E				ROL X	62
63	3F	?	?	?		63
64	40				RTI	64
65	41	A	,a	A	EOR(I,X)	65
66	42	B	,b	B		66
67	43	C	,c	C		67
68	44	D	,d	D		68
69	45	E	,e	E	EOR Z	69
70	46	F	,f	F	LSR Z	70
71	47	G	,g	G		71
72	48	H	,h	H	PHA	72
73	49	I	,i	I	EOR #	73
74	4A	J	,j	J	LSR A	74
75	4B	K	,k	K		75
76	4C	L	,l	L	JMP	76
77	4D	M	,m	M	EOR	77
78	4E	N	,n	N	LSR	78
79	4F	O	,o	O		79
80	50	P	,p	P	BVC	80
81	51	Q	,q	Q	EOR(I),Y	81
82	52	R	,r	R		82
83	53	S	,s	S		83
84	54	T	t,	T		84
85	55	U	u,	U	EOR Z,X	85
86	56	V	,v	V	LSR Z,X	86
87	57	W	,w	W		87
88	58	X	,x	X	CLI	88
89	59	Y	,y	Y	EOR Y	89
90	5A	Z	,z	Z		90
91	5B					91
92	5C					92
93	5D				EOR X	93
94	5E		↑		LSR X	94
95	5F		←			95
96	60				RTS	96
97	61				ADC(I,X)	97
98	62					98
99	63					99



DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL
100	64					100
101	65				ADC Z	101
102	66				RCR Z	102
103	67					103
104	68				PLA	104
105	69				ADC #	105
106	6A				RCR A	106
107	6B					107
108	6C				JMP(I)	108
109	6D				ADC	109
110	6E				ROR	110
111	6F					111
112	70				BVS	112
113	71				ADC(I),Y	113
114	72					114
115	73					115
116	74					116
117	75				ADC Z,X	117
118	76				ROR Z,X	118
119	77					119
120	78				SEI	120
121	79				ADC Y	121
122	7A		,			122
123	7B					123
124	7C					124
125	7D				ADC X	125
126	7E				ROR X	126
127	7F					127
128	80		r-O	END		128
129	81		r-A	FOR	STA(I,X)	129
130	82		r-B	NEXT		130
131	83		r-C	DATA		131
132	84		r-D	INPUT #	STY Z	132
133	85		r-E	INPUT	STA Z	133
134	86		r-F	DIM	STX Z	134
135	87		r-G	READ		135
136	88		r-H	LET	DEY	136
137	89		r-I	GOTO		137
138	8A		r-J	RUN	TXA	138
139	8B		r-K	IF		139
140	8C		r-L	RESTORE	STY	140
141	8D	car ret	r-M	GOSUB	STA	141
142	8E		r-N	RETURN	STX	142
143	8F		r-O	REM		143
144	90		r-P	STOP	BCC	144
145	91	cur up	r-Q	ON	STA(I),Y	145
146	92	rvs off	r-R	WAIT		146
147	93	clear	r-S	LOAD		147
148	94	insert	r-T	SAVE	STY Z,X	148
149	95		r-U	VERIFY	STA Z,X	149

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL
150	96		r-V	DEF	STX Z,Y	150
151	97		r-W	POKE		151
152	98		r-X	PRINT #	TYA	152
153	99		r-Y	PRINT	STA Y	153
154	9A		r-Z	CONT	TXS	154
155	9B		r-[	LIST		155
156	9C		r-\	CLR		156
157	9D	cur left	r-]	CMD	STA X	157
158	9E		r-↑	SYS		158
159	9F		r-←	OPEN		159
160	A0			CLOSE	LDY #	160
161	A1		r-!	GET	LDA(I,X)	161
162	A2		r-"	NEW	LDX #	162
163	A3		r-#	TAB(		163
164	A4		r-$	TO	LDY Z	164
165	A5		r-%	FN	LDA Z	165
166	A6		r-&	SPC(	LDX Z	166
167	A7		r-'	THEN		167
168	A8		r-(	NOT	TAY	168
169	A9		r-)	STEP	LDA #	169
170	AA		r-*	+	TAX	170
171	AB		r-+	-		171
172	AC		r-,	*	LDY	172
173	AD		r--	/	LDA	173
174	AE		r-.		LDX	174
175	AF		r-/	AND		175
176	B0		r-0	OR	BCS	176
177	B1		r-1		LDA(I),Y	177
178	B2		r-2	=		178
179	B3		r-3			179
180	B4		r-4	SGN	LDY Z,X	180
181	B5		r-5	INT	LDA Z,X	181
182	B6		r-6	ABS	LDX Z,Y	182
183	B7		r-7	USR		183
184	B8		r-8	FRE	CLV	184
185	B9		r-9	POS	LDA Y	185
186	BA		r-:	SQR	TSX	186
187	BB		r-;	RND		187
188	BC		r-	LOG	LDY X	188
189	BD		r-=	EXP	LDA X	189
190	BE		r-	COS	LDX Y	190
191	BF		r-?	SIN		191
192	C0			TAN	CPY #	192
193	C1		,a	ATN	CMP(I),X	193
194	C2		,b	PEEK		194
195	C3		,c	LEN		195
196	C4		,d	STR$	CPY Z	196
197	C5		,e	VAL	CMP Z	197
198	C6		,f	ASC	DEC Z	198
199	C7		,g	CHR$		199

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL
200	C8	,h		LEFT$	INY	200
201	C9	,i		RIGHT$	CMP #	201
202	CA	,j		MID$	DEX	202
203	CB	,k				203
204	CC	,l			CYP	204
205	CD	,m			CMP	205
206	CE	,n			DEC	206
207	CF	,o				207
208	DO	,p			BNE	208
209	D1	,q			CMP(I),Y	209
210	D2	,r				210
211	D3	,s				211
212	D4	,t				212
213	D5	,u			CMP Z,X	213
214	D6	,v			DEC Z,X	214
215	D7	,w				215
216	D8	,x			CLD	216
217	D9	,y			CMP Y	217
218	DA	,z				218
219	DB					219
220	DC					220
221	DD				CMP X	221
222	DE	,			DEC X	222
223	DF	,				223
224	EO				CPX #	224
225	E1				SBC(I),X	225
226	E2					226
227	E3					227
228	E4				CPX Z	228
229	E5				SBC Z	229
230	E6				INC Z	230
231	E7					231
232	E8				INX	232
233	E9		,		SBC #	233
234	EA				NOP	234
235	EB					235
236	EC				CPX	236
237	ED				SBC	237
238	EE				INC	238
239	EF					239
240	FO				BEQ	240
241	F1				SBC(I),Y	241
242	F2					242
243	F3					243
244	F4					244
245	F5				SBC Z,X	245
246	F6				INC Z,X	246
247	F7					247
248	F8				SED	248
249	F9				SBC Y	249

Reverse of ASCII

DECIMAL	HEX	ASCII	SCREEN	BASIC	6502	DECIMAL
250	FA		,			250
251	FB					251
252	FC					252
253	FD				SBC X	253
254	FE				INC X	254
255	FF			π		255

INDEX