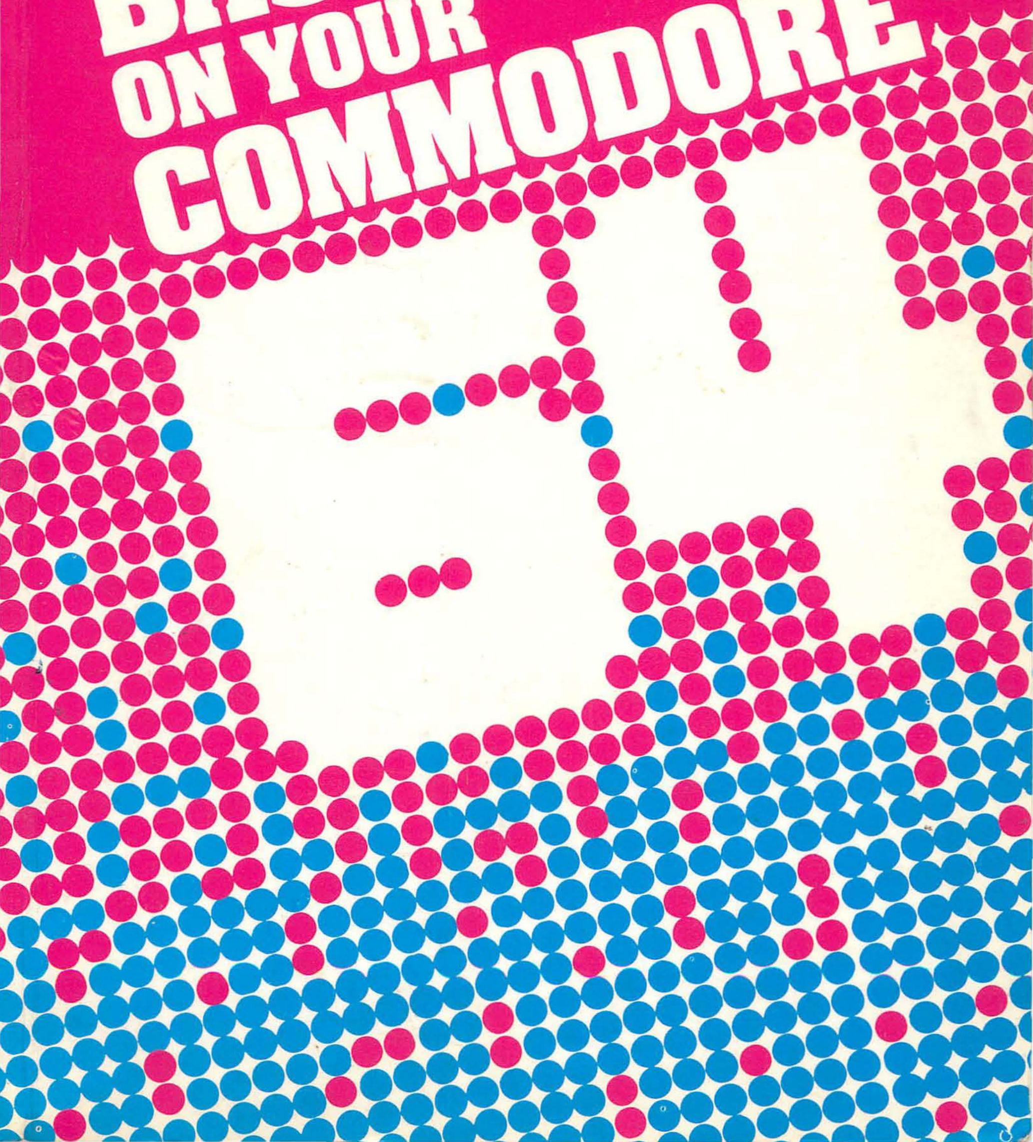
SIGMA PRESS

C.I. BURKINSHAW

BEYOND BASIC ON YOUR COMMODORE



BEYOND BASIC ON THE COMMODORE 64

C.I. BURKINSHAW

Σ
SIGMA
PRESS

Copyright © 1984 by C.I. Burkinshaw

All Rights Reserved

No part of this book may be reproduced or transmitted by any means without the prior permission of the publisher. The only exceptions are for the purposes of review, or as provided for by the Copyright (Photocopying) Act or in order to enter the programs herein onto a computer for the sole use of the purchaser of this book.

ISBN 0 905104 91 9

Typesetting and Production by:

DESIGNED PUBLICATIONS LTD.
8-10 Trafford Road,
Alderley Edge, Cheshire.

Published by:

SIGMA PRESS
5 Alton Road,
Wilmslow,
Cheshire.
UK.

Distributors:

Europe, Africa:
JOHN WILEY & SONS LIMITED,
Baffins Lane, Chichester,
West Sussex, England.

Australia, New Zealand, South-East Asia:
Jacaranda-Wiley Ltd., Jacaranda Press,
JOHN WILEY & SONS INC.,
GPO Box 859, Brisbane,
Queensland 40001, Australia.

Printed and bound in Great Britain by
J. W. Arrowsmith Ltd., Bristol

ACKNOWLEDGEMENTS

The author would like to thank G.T.C (Systems) Ltd of Warren St., Sheffield for the provision of word processing facilities and Commodore Business Machines (UK) Ltd for permission to use material from the 'The Commodore 64 Programmer's Reference Guide.

Special thanks to my family for their help and patience during the manuscripts preparation.

"Commodore 64" is a registered trade mark of Commodore Business Machines.

PREFACE

After all those hours spent learning BASIC and delving into the 64's graphics and sound, you find that for many applications it all happens too slowly.

Perhaps you've heard of machine language programming but thought it was only for the experts.

This book is written to allow the 64 user to easily obtain the maximum power from the computer by programming the 6510 microprocessor directly in machine code. In addition to widening the range of programs possible, it brings you closer to the inner workings of the computer.

In eight chapters you are shown how the 64 functions, where machine code fits in and how to exploit to the full the many facilities it offers.

Machine code is no more difficult to learn than BASIC but it is more difficult to use effectively.

Whereas BASIC was designed to allow the user to program with little knowledge of the machine, a more detailed knowledge of its function is necessary for machine code programming.

Consequently, as this book introduces machine code, it does so in tandem with sufficient information to allow the competent BASIC programmer to use the new skills productively.

CONTENTS

1.	**Preliminaries**	1
	Microprocessors	1
	High Level Languages	3
	Binary and Hexadecimal Representation	4
	Logical Operations	6
2.	**The 6510 Processor**	9
	6510 Architecture	11
	Memory Usage	14
3.	**Beginning Machine Code**	17
	Instruction Framework	17
	Memory Movement	18
	Increment and Decrement Functions	19
	Arithmetic Operations	19
	Logic Operations	20
	Comparison and Branch Instructions	21
	Subroutines	23
	Example Program	24
	The 6510 Instruction Set	25
4.	**Addressing Techniques**	53
	Literals and Absolutes	53
	Implied	54
	Immediate	54
	Absolute	54
	Zero Page	54
	Zero Page-X, Zero Page-Y	55
	Absolute X	55
	Absolute Y	55
	Indirect X	55
	Indirect Y	56
	Binary Coded Decimal	58
	Floating Point	59
	Multiple Precision	60
	The Overflow Flag	60

5. **Stacks and Subroutines** 63
More on Interrupts 63
Stack Instructions 64
Subroutine Use 65
Program Structure 66
Sorting 66
Positional Independence 67

6. **Screen and Keyboard Techniques** 69
The CIA 1 69
The Interrupt Routines 72
SCNKEY and Associates 73
Keyboard Handling 74
The Function Keys 75
Sprites and Animation 76
Writing and Using Larger Programs 77
A Hex Loader 77
An Assembler 78
Graphics and the 64 80
Low Resolution Mode 86
Example Program "Text Screen" 87
Bit Map Mode 91
Multicolour Mode 93
Scrolling 93
Split Screen Effects 94
Example Program "Mine Shaft" 95
Example Program "Prong" 102

7. **The Commodore 64 ROM** 109
Overview 109
Program Storage 109
Variables 110
Input/Output 112
Joysticks 112
The User Port 113
Serial Bus 113
Finishing Off 114
Other Kernal Routines 114
The System Clock 114
Screen Routines 115
System Procedures 115

8. Odds and Ends 117
Cassette Handling 117
Files 118
Program Protection 119
More Resident Routines 121

Appendix A : Decimal - Hexadecimal 125
Appendix B : Colour Table 126
Appendix C : 6510 Instructions 127
Appendix D : Kernal Routines 128
Appendix E : Memory Map 129
Appendix F : Sprite Table 134
Appendix G : VIC Registers 135
Appendix H : CIA 1 and CIA 2 137
Appendix I : I/O Error Messages / Device Status 140

CHAPTER 1

PRELIMINARIES

Microprocessors. High level languages. Binary and Hexadecimal Representation. Logical Operations.

Microprocessors

At the heart of any small computer lies a microprocessor.
This unit is programmed directly using machine code and has three main capabilities:

i. The movement of data between locations.
ii. Basic arithmetic.
iii. Logic operations.

In addition, 8 bit processors only allow data in the range 0-255 to be manipulated directly (the binary number system uses 8 bits to represent values upto 255 decimal). Operations on larger data require division into stages. As no specific instructions are available for this, the programmer must provide the necessary routines. Clearly, the larger the data that a processor may manipulate directly, the more efficient the processing. This is the principal advantage of the more recent 16 and 32 bit designs.

As we mentioned, apart from simple arithmetic, the processor's main ability is the movement of data between locations. The size of memory that may be addressed is limited, for the 64's 8 bit 6510, the largest address allowed is 65535. At this stage it may be helpful to visualize the memory as an array of pigeon-holes, each capable of holding a number in the range 0-255.

This array is external to the processor and may consist of Read Only Memory (R.O.M.) or Random Access Memory (R.A.M.). The latter may be used to either store or read data, whilst R.O.M. may only be read. In dealing with a memory area of this size it is convenient to break it down into manageable blocks. A common division is into pages, each holding 256 addresses. The opening page is known as page zero and contains locations 0 to 255, consequently the first page boundary occurs after address 255.

Instructions to the microprocessor take the form of a number between 0 and 255. If an instruction requires a data location to be specified then it should precede the next instruction, and be either one or two bytes in length. A sub-set of the instruction menu is only for operations involving zero page locations. As these execute more quickly than their general memory equivalents, almost all small computers reserve zero page for system use.

Certain internal memory locations on the processor are accessible to the machine code programmer. The most useful include:

1. Process Status Register (P.S.R.)
2. Program Counter
3. Stack pointer
4. The general purpose registers

All are 8 bits wide except the program counter which is 16. Plainly, the processor must know where its instructions are located, and this is the function of the program counter register, which holds the address of the next instruction.

The process status register contains information on both systems status and the parameters of the most recent instruction executed, e.g. if a subtraction resulted in a negative value, the left-most bit or flag would be set. The P.S.R. is used by the branch options in the instruction menu. These will reset the program counter to point to a new location if one of the P.S.R. bits holds a particular value.

Last and most important, are the general purpose registers. The 6510 has three:

i. Accumulator
ii. X register
iii. Y register

Their importance stems from the system architecture of the 6510. To perform addition or subtraction, the processor requires one of the values

to be placed in the accumulator.

After execution the result is deposited in the accumulator.

Although this allows rapid evaluation, a major disadvantage is the use of considerable programming space to effect movement of data to and from the accumulator.

The branch instructions mentioned provide the decision options available to the machine code programmer. In BASIC a branch is to a specified line if certain conditions are satisfied. There is no equivalent in machine code. Instead, the branch must be to an address within either 127 locations forward or 128 back from that replaced.

An absolute jump instruction is available to transfer control to any addressable location. As with the equivalent BASIC command, GOTO, it cannot be made conditional.

High level languages

Held on a R.O.M. chip in the 64 is a machine code program called the operating system. This provides and co-ordinates functions within the computer. On receipt of a BASIC command, the operating system converts it into a series of machine code instructions. This conversion or compilation is performed by the other resident machine code program - the BASIC interpreter. On the 64 the operating system is between locations 57344 and 65535, while the BASIC interpreter extends from 40906 to 49151.

The 64's memory has an unusual configuration, consisting of a 'base layer' of 64K of R.A.M. which is overlaid by 20K of R.O.M., any attempt to read the 20K of R.A.M. will result in a R.O.M. access.

In designing the machine, Commodore allow the user to switch out sections of R.O.M. Consequently, whenever only machine code programs are to be run, which require neither the operating system nor BASIC interpreter, then the underlying 16K of R.A.M. may be utilized.

This illustrates two of the principal advantages of machine code over the use of a high level language like BASIC:

i. Extra memory is available for program use.
ii. The available program space may be used more economically due to the greater purpose-specifity of machine code.

Reference has been made to BASIC as a 'high level language'. Just what is a 'high level language? The answer is simple, it is one that supports commands that are not intrinsic to the microprocessor. As a result, when a high level language is used, an interpreter program is necessary. The time taken to execute an instruction is the amount of time taken to translate the original command into machine code and then to execute it. Eureka! The attraction of machine code - speed of execution.

Binary & Hexadecimal Representation

Microprocessors were not designed to use the decimal number system, and to be able to program in machine code it is necessary to be fluent in both the Hexadecimal & Binary number systems.

In decimal there are ten possible digits 0-9, in binary there are only two, 0 & 1, so any binary number is a string of 1's and 0's. The other main difference is that instead of each column representing a value ten times that of the column to its right, in binary it is double.

Decimal	*Binary Representation*							
	128	64	32	16	8	4	2	1
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	1	1
4	0	0	0	0	0	1	0	0
5	0	0	0	0	0	1	0	1
6	0	0	0	0	0	1	1	0
7	0	0	0	0	0	1	1	1
8	0	0	0	0	1	0	0	0
9	0	0	0	0	1	0	0	1
100	0	1	1	0	0	1	0	0
200	1	1	0	0	1	0	0	0

Figure 1.1 Binary column values

To convert a binary number to the decimal equivalent, it is necessary to add the values represented by the columns in which the number has 1's. In figure 1.1 the binary equivalent of 100 decimal is seen as 64 + 32 + 4. Each 1 or 0 is referred to as a bit, a number with eight bits being termed a byte. Clearly, the largest number that can be represented by an eight bit number is 128+64+32+16+8+4+2+1=255. It is conventional to include 0's to the full eight bits for those values less than 255.

The right hand bit is often called the Least Significant Bit (L.S.B.) and the leftmost bit the Most Significant Bit (M.S.B.). Addition and subtraction are straightforward with one exception - the representation of a negative number. As no equivalent of the minus sign is available, negative numbers are formed in a different manner. The method of coding used is called two's complement notation. This is obtained by taking the absolute value of the negative number and converting that to its binary form.

Each digit is then inverted, and finally 1 is added.

For example, to obtain the two's complement of -12, take the binary representation of its absolute value (12) 00001100. Invert it 11110011, and then add 1 = 11110100. As the computer must be able to differentiate this from 244 decimal, which has the same representation, a limit is imposed on the range of values, from -128 to 127. This prevents any overlap occurring.

Operation	*Example*
Negative value	-121
Absolute value	121 / 01111001
Inverted form	10000110
Increment	00000001
Two's complement	10000111

Figure 1.2 Representation of negative values in binary

Any eight bit number may be split to give two four bit 'nibbles' (Half a byte). In isolation, each represents a value in the range 0-15. This is the basis of a useful shorthand for binary data. Clearly, 16 different digits are needed, and 0-9 are supplemented with the first letters of the alphabet representing the values 10-15.

Decimal	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
Hexadecimal	0 1 2 3 4 5 6 7 8 9 A B C D E F 10 11

Example: 56 decimal = 00111000 binary.
The nibbles are: 0011 and 1000, values 3 and 8
The Hexadecimal form is 38

Figure 1.3 Hexadecimal notation

This method of representation is the Hexadecimal number system. The conversion illustrated in Figure 1.3 shows the simplicity achieved over binary groups. To convert two digit hexadecimal values to decimal just multiply the left digit by 16 and add it to the righthand value.

You may have noticed that Hexadecimal is a number system to the base 16. To convert from decimal to Hexadecimal, it is only necessary to break the value down into the component powers of 16.

Hexadecimal number	4	3	A	4
Column Values	4096	256	16	Actual
Evaluation	4*4096 +	3*256 +	10*16 +	4 = 17316

Figure 1.4 Hexadecimal conversion to decimal notation

It is conventional to indicate the hexadecimal form either by a suffix of 'H' or a prefix of '&'. the latter correction is used in this book.

Logical Operations

Consider the following problem: Given two 8 bit binary numbers, you are required to form another byte with 1's in only those positions in which both the original bytes also have 1's.

The answer? Simple: first, one value is loaded into the accumulator, second, the processor performs a logical AND operation between the

other value and the accumulator. The result - the byte required - is found in the accumulator.

The AND operation is one of three logical operations the 6510 will perform, which compare each bit of two bytes, and depending on the basis of that comparison, form a new value in the accumulator.

As we have seen, the AND operation will place a 0 in each bit position unless both comparison data also contain a 1 in that position. The inverse operation to this is the ORA command, which only returns a 1 in any position if the initial data contains 0's. The third instruction, EOR, is a cross between the previous two, it requires both data to have dissimilar digits in order that a 1 result in the accumulator.

To the newcomer to machine code, these operations may seem unimportant - not so, they are often required for certain calculations and the control of particular functions.

Two examples:

i. To obtain the two's complement of a positive number, load the accumulator with -1, and then perform the EOR operation between it and the positive value.

ii. The operation of masking. This is performed to ascertain if a byte has a particular bit set. The value is AND'd with a test byte with only the relevant bit set. This masks away any other bytes that may have be set. If the result is zero then the original byte holds a 0 in the test position.

Operation:	*AND*	*EOR*	*ORA*
Condition for 1 to result in a particular column	Both data have a 1 in the column	Both data have a 0 in the column	Data have dissimilar digits

Figure 1.5 The 6510 Logic Operations

The next four chapters deal with the format and use of the 6510 and its instruction set. Later chapters then cover their use on the 64 and demonstrate how to access the resident machine code programs - the BASIC interpreter and operating system.

CHAPTER 2

THE 6510 MICROPROCESSOR

System Organisation. The 6510 Architecture. Memory Usage. Input / Output Operations.

The 6510 microprocessor is one of a family of integrated packages that together provide several of the central facilities of a microcomputer. In the 64 there are four main units. Two of these are 6526 Complex Interface Adapters (C.I.A.s), and are concerned with input and output. The sound facilities emanate from the 6581 Sound Interface Device (S.I.D.) and the video output is looked after by the 6569 Video Interface Chip (V.I.C. 2).

Communication between these units and the C.P.U. occurs via the address and data buses. The address bus consists of 16 lines or wires, each of which carries one bit of an address. As data is handled by the processor in bytes, the data bus need only be 8 lines wide.

Plainly, the system needs to be exactly synchronised and a timing reference is available via the V.I.C. 2 chip. This 8MHz clock is stepped down to 1 million cycles per second for system use. An indication of the tempo of the system is that an instruction to load a value from memory into the accumulator executes in 2-6 cycles. It also becomes clear why it is often difficult to identify program errors!
In the normal operating state, the control registers of the main components appear to the microprocessor as memory locations and as a result are said to be 'memory mapped'. This permits most input/output operations to be initiated by the C.P.U. writing values to the appropriate locations.

To gain the attention of the processor, in order to carry out an urgent function, a component may cause an 'interrupt'. The 6510 then completes the current instruction and branches to the address held in locations 65534 and 65535 (IRQ). The program pointed to determines which device initiated the interrupt, and the appropriate action is taken.

On completion, control reverts to the original program.

It is often not necessary to use the interrupt procedure if only use of the address and data buses are required. The V.I.C. 2 chip reviews sections of memory each microsecond, to form its output. To do this it uses a technique known as Direct Memory Access (D.M.A.). As we mentioned, the system clock operates at 1 MHz, with each cycle consisting of several phases. The processor will only utilize the address and data buses during phase 2, leaving them clear during phase 1. The V.I.C. 2 package takes advantage of this to perform a refresh operation on both the 64's dynamic R.A.M. and the memory access.

Figure 2.1 gives an overview of the system design, but as several components rival the 6510 in complexity, a seperate volume would be needed to consider their functions in depth.

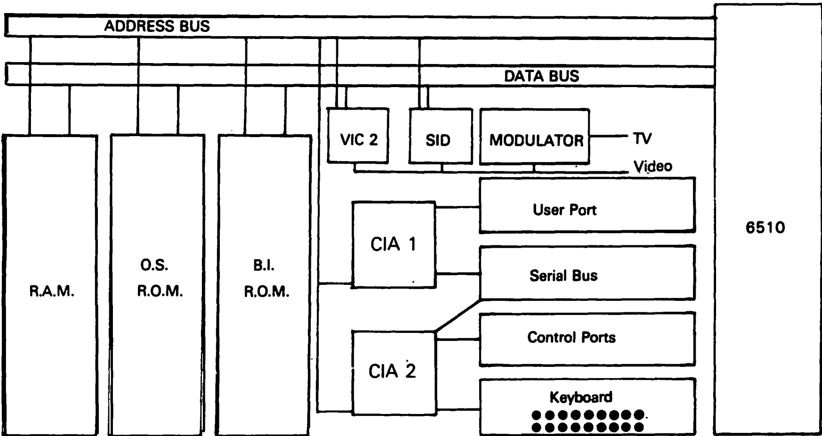


Figure 2.1 System outline

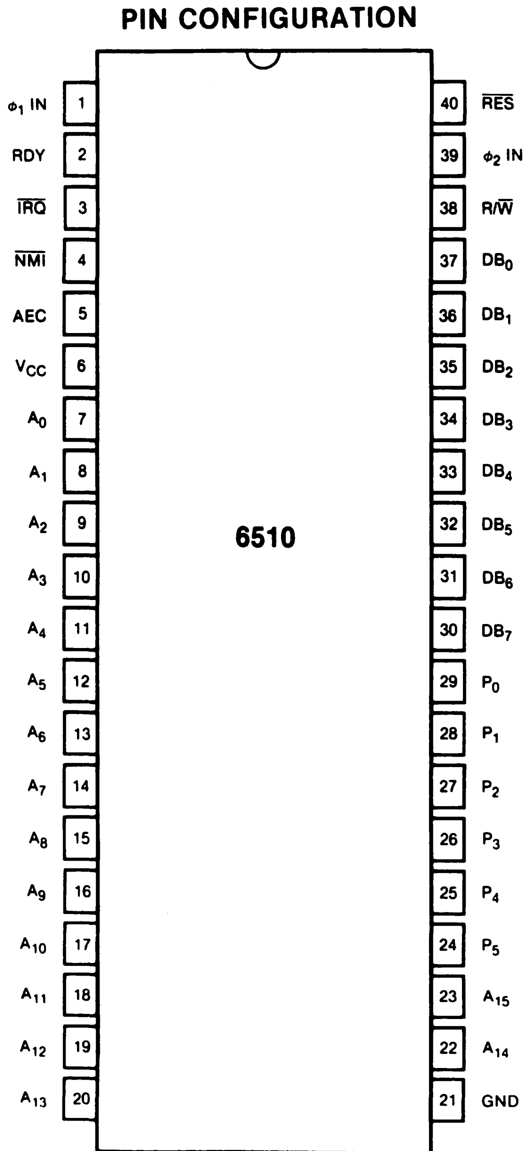


Figure 2.2 The 6510 Pin Configuration

The 6510 is manufactured by MOS Technology (a subsidiary of Commodore Business Machines). It operates from a single 5 volt supply. In structural configuration it is a typical Dual In Line Integrated Circuit (D.I.L.I.C.) with 40 pins. The address bus requires 16 lines and the data bus eight. The remaining sixteen pins break down into:

Input / Output port	7	Reset line	1
Interrupt facility	2	Read / Write line	1
Power supply	2	Address enable line	1
Clock input lines	2		

The Read / Write line indicates to R.A.M. which operation the C.P.U. intends to perform, and the Address enable line if the C.P.U. is isolated from the address bus.

Internally the 6510 comprises an Arithmetic Logic Unit (A.L.U.), control unit, registers and Read Only Memory. Their operation is best shown by looking at a sample operation -

i. *The address* pointed to by the program counter register is accessed and loaded into a second register in the processor. This instructs the processor to load the accumulator register with the byte in the next location.
ii. *The C.P.U.* increments the program counter register to point to the next instruction.
iii. *The address* of the *data byte* is put on the address bus.
iv. When the byte is received onthe data bus it is stored in the accumulator. This overwrites any previous value held.
v. If necessary *flag bits* in the P.S.R. (Processor Status Register) are set.

The entire operation completes in two clock cycles. The X and Y registers may also be directly loaded with data in a similar manner.

However, the three registers are frequently inadequate for storing all the information we need on hand. Happily, the 6510 provides a larger store - the Stack. Any value held in the accumulator may be 'pushed' onto the Stack, and at a later stage it may be 'pulled' back into the accumulator.

The stack is not limited to holding a single value, but will expand to hold a total of 255 bytes. The drawback attached to stack use is that values may only be retrieved in the reverse order to which they were 'pushed'. It is also worth mentioning that the value pushed onto the stack remains in the accumulator, so to examine the top value, the most economical method is to pull and then push it back.

The stack uses page two of R.A.M. for byte storage (locations 256-511), and it is inadvisable to use this space for your own routines (This applies even if the routine does not use the stack, as the C.P.U. may do).

An analogy often made to the stack, for illustrative purposes, is a pile of plates. This is misleading because as values are added, the stack fills 'up' from location 511 downwards..

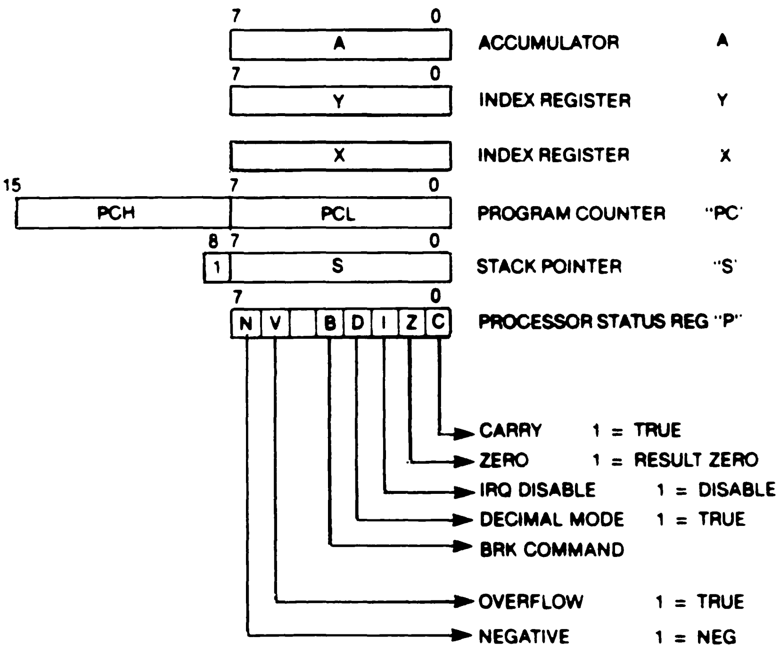


Figure 2.3 Processor Model

The stack pointer register normally holds the address of the next free byte from the 'top' of the stack, and is programmable.

Memory Usage

Locations		*Content*	
65535 - 57344	*	Operating System	8K
57343 - 57088	*	Disk / O	256
57087 - 56832	*	CP / M	256
56831 - 56576	*	C.I.A. 2 Registers + images	256
56575 - 56320	*	C.I.A. 1 Registers + images	256
56319 - 55296	*	Screen colour attributes + images	1K
55295 - 54272	*	S.I.D. control registers + images	1K
54271 - 53248	*	V.I.C. 2 control registers + images	1K
53247 - 49152		R.A.M. buffer	4K
49151 - 40960	*	BASIC interpreter	8K
40959 - 2048		User memory	38K
2047 - 2040		Pointers to sprite definition	48
2039 - 1024		Screen character map	1K
1023 - 2		Reserved for system use	1K
0 & 1		I / O Port & direction	2

* Indicates R.O.M. which may be banked out.

Table 2.1 Commodore 64 Memory Map

Location 1 is used by the 6510 as an Input / Output port, which, in conjunction with location 0 enables blocks of R.O.M. to be replaced by the underlying R.A.M. in the microprocessor's memory map. Each bit in address zero indicates whether the equivalent bit in location 1 is for input or output (a setting of 1 = output).

After power up, this contains 47 decimal with bits 0-3 set high. if the Least Significant Bit of location 1 is brought 0 low the R.O.M. holding the BASIC interpreter is switched out - bit 1 as an equivalent effect on the R.O.M. holding the operating system. Setting bit 2 low substitutes the character R.O.M. for the Input / Output configuration.

To communicate with devices that do not form part of the base system, the 64 may use the expansion port, serial bus or user port. Irrespective of which is selected, the data transference is either serial or parallel in format. Serial I / O requires that the data is transferred a bit at a time.

Using this method, the transmission of a byte involves 8 pulses. Parallel data transfer requires 8 lines, one for each bit, and a complete byte may be transferred concomitantly.

Both methods rely upon exact timing between devices.

This is often accomplished by the process of 'Handshaking'. Here, data transmission is prefaced with control signals. These may need to be acknowledged, as might subsequent data sections.

There are two approaches to overseeing this type of operation; either complex software may be developed or the functions may be supervised by dedicated hardware. A software solution involves permanent memory space and processing time, whilst the hardware option forces a higher original cost.

The 6526 package represents a low cost compromise, with the C.I.A. 1 unit primarily concerned with keyboard data whilst C.I.A. 2 looks after the User port & Serial bus. Tape I/O is handled via the 6510's on-board port.

CHAPTER 3

BEGINNING MACHINE CODE

Instruction framework. Comparison & Branch instructions. Example Program. Instruction set.

Instruction framework

Unlike a BASIC program, which is a set of distinct lines, a machine code routine consists of an unbroken sequence of instruction and address bytes.

Each of the 151 instructions the processor will execute is specified by an opcode one byte in length. If a data address is necessary it follows the instruction and is one or two bytes in length.

Memory location	*Contents*	*Function*		
3006	& 0E	Opcode	:	ASL
3007	& 20	Address		
3008	& 34	Address		
3009	& 18	Opcode	:	CLC
3010	& 4C	Opcode	:	JMP
3011	& FF	Address		
3012	& AA	Address		

Figure 3.1 A segment of machine code

At this level, unless an assembler program is used, it is normal to work in the hexadecimal notation.

An assembler will permit code to be entered, not as a series of bytes, but *via* mnemonics representing both instructions and addrsses. When the listing is complete the mnemonics are converted to the appropriate hexadecimal values.

If attempting to write programs of any length, an assembler provides the very necessary speed and accuracy.

It is important to note that the facilities offered by commercially available assembler programs vary considerably. The less complex versions - 'single pass' assemblers will, if the mnemonics contain branches to points further down the listing which the assembler has not yet encountered, leave a space. The address is then inserted when the label is encounted. A multi-pass version will creat a list of labels on the first run for insertion on a second pass.

Although the number of instructions available on the 6510 is large, many perform the same function. The duplication occurs to allow each operation to be used with the different address formats.

The popularity of the 6500 series of processor is, in part, due to the flexibility these afford. A drawback is the associated complexity. Chapter 4 has a closer look at the available address modes.

The instruction set has six main categories which we will look at first.

i. Memory movement
ii. Increment and decrement functions
iii. Arithmetic operations
iv. Logic functions
v. Comparison and Branch Instructions
vi. Subroutines

Memory movement

These divide into two types, which either:

1. Store from a register to general memory.
2. Load to a register from memory.

Each type has 3 opcodes, each of which is specific to a general register:

LDA, LDX, LDY (LoaD register) and STA, STY, STX (STore from the register).

It must be stressed that the source byte is unchanged. For example, the opcode &8D causes the value held in the two byte address following the instruction to be placed in the accumulator.

Opcode &85 performs the same operation except that the address only occupies one byte. The source therefore lies in the zero page.

Increment and decrement functions

These either decrease or increase the value in the address or register specified, by 1.

As the X and Y registers are often used as counters in repetitive operations, they are termed the index registers.

To facilitate this action they possess special increment and decrement instructions.

Arithmetical operations

Only one instruction will perform addition - the ADC (ADd with Carry) operation. This adds the contents of the accumulator to the data specified. The result is placed in the accumulator.

If the carry flag is set, then 1 is also added to the result. This facilitates the addition of values in excess of 255, by providing for a carry from one stage to the next. Plainly, prior to ordinary addition, the carry flag should be cleared.

Use the CLC (CLear Carry) instruction for this.

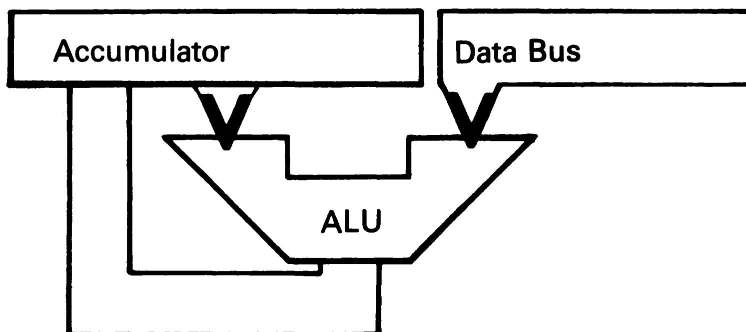


Figure 3.2 The add with carry operation

To perform subtraction the SBC (SuBtract with Carry) instruction is used. This decrements the value in the accumulator by the data specified. Unless a borrow from a previous calculation is to be taken into account, the carry flag must be set prior to execution with the SEC (SEt Carry) instruction.

The reader will have appreciated that if each bit in a binary number is moved one position left, the value represented is doubled. The bit position shift forms the basis of the processor's division and multiplication operations.

The ASL (Arithmetic Shift Left) instruction displaces a byte one position left. The M.S.B. is not lost, but placed in the carry bit. The L.S.B. is set low. The complementary instruction, LSR (Logical Shift Right), sets the M.S.B. low, and places the L.S.B. in the carry bit.

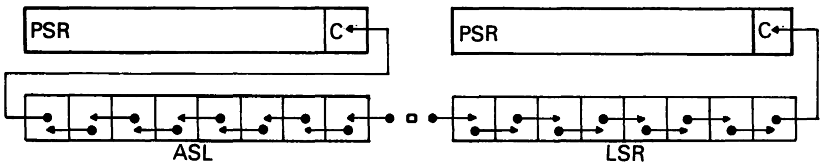


Figure 3.3 The ASL and LSR operations

Logic functions

Two of the logic functions, ROL (ROtate Left) and ROR (ROtate Right), are similar to the shift operations. The difference lies in the treatment of the vacant bit. Instead of being set low, it is loaded with the carry bit, which, in turn is loaded with the displaced M.S.B.

The remaining logic function, BIT, performs two operations. First, bits 6 and 7 of the data are placed in the V and N positions of the P.S.R. Second, the logical AND is performed between the accumulator and the data. However, the result is not placed in the accumulator, but is used to update the Z flag. If both bytes are the same the flag is set, otherwise it is cleared.

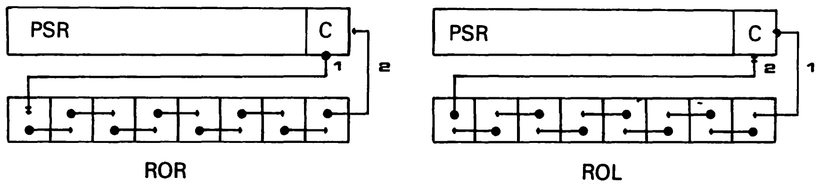


Figure 3.4 The ROR and ROL operations

The Comparison and Branch instructions

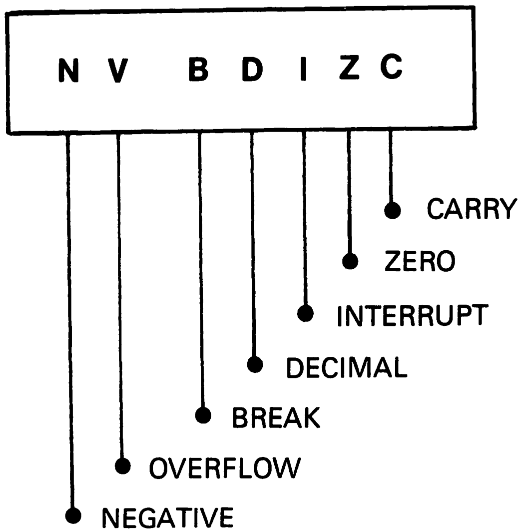


Figure 3.5 The Process Status Register (PSR)

If you want to branch to another part of the program, but only if a condition is satisfied, one of the eight branch instructions must be used. In BASIC a combination of conditions is possible with the IF-THEN statements. However, in machine code you are limited to whether one of the P.S.R. flags is set or not.

The length of the jump is held in the byte following the instruction. A reverse jump is implemented by the use of the two's complement notation. This places a limit on the maximum displacement to either 128 bytes back or 127 forwards.

For example, a jump 50 locations forward on a result of zero may be implemented as follows:

i. Load the accumulator with the result. If this is zero, the Z flag is set.
ii. A BEQ (Branch if zero) is then used to initiate the jump. The opcode, &FO, precedes the displacement, &32.
iii. The displacement is added to the address held in the program counter to give the location of the next instruction to be executed.

Figure 3.6 below, illustrates the event sequence.

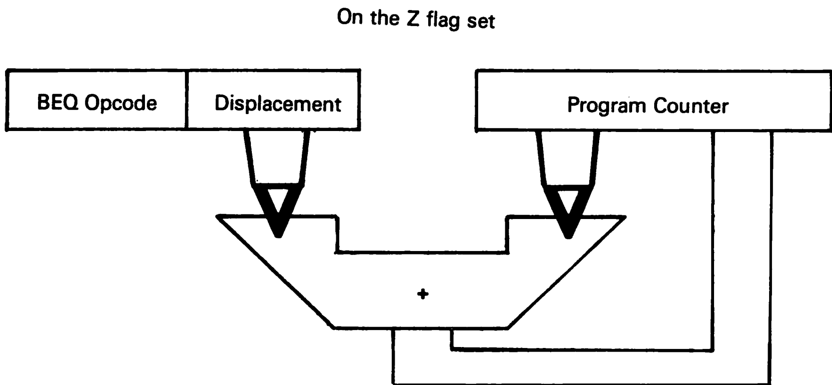


Figure 3.6 The BEQ instruction

Branch instructions are often a source of error in machine code programming, with programmers happily re-directing control to locations 1 or 2 positions away from that intended. It is important to remember, in calculating a negative displacement, that the Program Counter (P.C.) holds the address of the next instruction, so the displacement should also include the two bytes comprising the branch instruction.

Mnemonic	*Condition*	*Opcode*
BCC	Carry not set	&90
BCS	Carry set	&B0
BMI	Negative flag set	&30
BPL	Neg. flag not set	&10
BNE	Zero flag clear	&D0
BVC	V flag clear	&50
BVS	V flag set	&70

Table 3.1 The Relative branch options

In isolation, the branch instructions do not permit a significant degree of flexibility. A second group, the Compare instructions, allow a wider range of conditions to initiate a jump. There are three types, each specific to a general register: CPX, CPY and CMP. These cause the value held in a given memory location to be subtracted from the register. The result updates the N, Z and C flags in the P.S.R. Importantly, neither comparison data is altered. A branch instruction will always follow the Compare operation.

As with most instruction types, the compare operation contains opcodes which do not expect a data address to follow the instruction, but the actual value.

For example, the opcode &C9 - CMP - sets the zero flag should the accumulator hold the same value as the byte after the instruction.

An unconditional jump to a new execution address is forced using the JMP instruction. The operation is not limited by a maximum displacement - the address specified is merely loaded into the program counter.

Subroutines

The equivalent of the BASIC GOSUB - RETURN structure is implemented with the JSR (Jump to SubRoutine) and RTS (ReTurn from Subroutine) instructions. The JSR call operates by placing the absolute address with the instruction in the program counter, after pushing the current address on the stack. The routine is ended if an RTS instruction is encounted. This pulls the old address from the stack, into the P.C.

Routines may be nested until the stack is full. However, as the operating system also utilizes the stack, it is unwise to approach the theoretical limit.

The RTS instruction has one other use - the BASIC SYS command. If the machine code routine contains an RTS without a prior JSR call, control reverts to BASIC.

Almost the entire instruction set has now been introduced. The main omissions relate to System Control and stack operations and are considered in chapter 5.

Example Program

We have now learnt sufficient to attempt our first program. It may be either poked into the locations given or the mnemonics assembled by an appropriate program.

The routine is then run using the command SYS 679 It will set the border colour to black and the screen colour to red. Notice that the opcode &A9 is the immediate variant of the LDA operation, so the actual data value, rather than an address, follows the instruction.

Several locations which will not be overwritten by BASIC are available on the 64 for machine code applications. For shorter routines use locations 679 to 767, otherwise the 4K block from 49152 to 53247 is best.

If it is not intended to use a BASIC program then there is no reason why locations 2048 to 40959 should not be utilized.

Address	*Hex. Code*	*Mnemonic*	*Comment*
679	A9	LDA	Load accumulator
680	0		with zero
681	8D	STA	Store accumulator
682	20		in &D020 (53280)
683	D0		
684	A9	LDA	Load accumulator
685	2		with 2
686	8D	STA	Store accumulator
687	21		in &D021 (53281)
688	D0		
689	60	RTS	Return from subroutine

The 6510 Instruction Set

The remainder of this chapter forms a reference section of the complete 6510 instruction set.

For each instruction we give the mnemonic (e.g. ADC), a description, the permitted addressing modes and, under the 'FLAGS' heading, the particular flags that can be modified by the instruction.

ADD WITH CARRY **ADC**

FLAGS	*Address Mode*		*Opcode*	*Bytes*
N V Z C	Immediate	....	69	2
	Absolute	....	6D	3
	0 - Page	....	65	2
	Abs. - X	....	7D	3
	Abs. - Y	....	79	3
	0 P. - X	....	75	2
	(Ind.- X)	....	61	2
	(Ind) - Y	....	71	2

DESCRIPTION

The data specified is added to the value held in the A. If the carry flag is set the result is incremented by 1.
The outcome is placed in the A.

LOGICAL AND **AND**

FLAGS	*Address Mode*		*Opcode*	*Bytes*
N Z	Immediate	....	29	2
	Absolute	....	2D	3
	0 - Page	....	25	2
	Abs. - X	....	3D	3
	Abs. - Y	....	39	3
	0 P. - X	....	35	2
	(Ind. - X)	....	21	2
	(Ind) - Y	....	31	2

DESCRIPTION

The logical AND is performed between the specified data and the accumulator. The outcome is placed in the accumulator.

ARITHMETIC SHIFT LEFT — ASL

FLAGS	*Address Mode*		*Opcode*	*Bytes*
N Z C				
	Immediate			
	Absolute	....	0E	3
	0 - Page	....	06	2
	Abs. - X	....	1E	3
	Abs. - Y			
	0 P. - X	....	16	2
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

Each data bit is shifted one position left. The L.S.B. is set low, and the M.S.B. is placed in the carry position.

BRANCH ON CARRY CLEAR — BCC

FLAGS	*Address Mode*		*Opcode*	*Bytes*
	Relative	....	90	2
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	Indirect			

DESCRIPTION

If the carry flag is clear, control is transfered to PC + the given displacement.

BRANCH ON CARRY SET — BCS

FLAGS	*Address Mode*		*Opcode*	*Bytes*
	Relative	....	B0	2
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	Indirect			

DESCRIPTION

If the carry flag is set, control is transfered to PC + displacement.

BRANCH IF DATA ZERO — BEQ

FLAGS	*Address Mode*		*Opcode*	*Bytes*
	Relative	....	F0	2
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	Indirect			

DESCRIPTION

If the Z flag is set, control is transfered to P.C. + the given displacement.

TEST ACCUMULATOR BITS WITH DATA — BIT

FLAGS	*Address Mode*		*Opcode*	*Bytes*
	Immediate			
	Absolute	....	2C	3
Z	0 - Page	....	24	2
	Abs. - X			
	Abs. - Y			
	O P. - X			
	Indirect			

DESCRIPTION

On the A. and data being equal the Z flag is set, otherwise cleared. Bits 6 and 7 of the data byte are placed in the N and V positions of the P.S.R.

BRANCH ON MINUS — BMI

FLAGS	*Address Mode*		*Opcode*	*Bytes*
	Relative	....	30	2
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	0 P. - X			
	Indirect			

DESCRIPTION

If the N flag is set, a branch is forced to P.C. + the given displacement.

BRANCH ON NON-ZERO RESULT — BNE

FLAGS	*Address Mode*		*Opcode*	*Bytes*
	Relative	....	D0	2
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	Indirect			

DESCRIPTION

On the Z flag being clear, a branch is forced to P.C. + displacement.

BRANCH ON PLUS — BPL

FLAGS	*Address Mode*		*Opcode*	*Bytes*
	Relative	....	10	2
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	Indirect			

DESCRIPTION

If the N flag is not set, a branch to the address held in the P.C. + the given displacement is executed.

BREAK — BRK

FLAGS	*Address Mode*		*Opcode*	*Bytes*
I B				
	Implied	....	00	1
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	Indirect			

DESCRIPTION

The software interrupt. Control is diverted to the address held in locations FFFE and FFFF. The address held in the P.C. + 2 is saved to the stack. When a RTI is encountered, control is passed back to this address.

BRANCH ON OVERFLOW CLEAR — BVC

FLAGS	*Address Mode*		*Opcode*	*Bytes*
	Relative	....	50	2
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	Indirect			

DESCRIPTION

On the V flag being clear, a branch to location P.C. + given displacement is implemented.

BRANCH ON OVERFLOW SET — BVS

FLAGS	Address Mode		Opcode	Bytes
	Relative	....	70	2
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	Indirect			

DESCRIPTION

On the V flag being set, a branch is executed to the location P.C. + displacement.

CLEAR CARRY FLAG — CLC

FLAGS	Address Mode		Opcode	Bytes
C				
	Implied	....	18	1
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	Indirect			

DESCRIPTION

The carry flag is set low.

EXIT FROM THE DECIMAL MODE — CLD

FLAGS	*Address Mode*		*Opcode*	*Bytes*
D				
	Implied	....	D8	1
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	Indirect			

DESCRIPTION

The Decimal flag is cleared. Arithmetical operations revert to the binary mode.

CLEAR INTERRUPT FLAG — CLI

FLAGS	*Address Mode*		*Opcode*	*Bytes*
I				
:				
	Implied	....	58	1
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	Indirect			

DESCRIPTION

The interrupt bit is set low. This enables the IRQ and BRK operators. The NMI is not affected by the interrupt flag.

CLEAR OVERFLOW FLAG — CLV

FLAGS	*Address Mode*		*Opcode*	*Bytes*
V				
	Implied	....	B8	1
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	Indirect			

DESCRIPTION

The V bit in the P.S.R. is set low.

COMPARE DATA TO ACCUMULATOR — CMP

FLAGS	*Address Mode*		*Opcode*	*Bytes*
N Z C				
	Immediate	....	C9	2
	Absolute	....	C9	3
	0 - Page	....	C5	2
	Abs. - X	....	DD	3
	Abs. - Y	....	D9	3
	0 P. - X	....	D5	2
	(Ind. - X)	....	C1	2
	(Ind) - Y	....	D1	2

DESCRIPTION

The specified data is subtracted from the accumulator.
Neither data is altered. The N, Z and C flags are updated from the result.

COMPARE DATA TO REGISTER X — CPX

FLAGS	Address Mode		Opcode	Bytes
N Z C				
	Implied			
	Immediate	....	E0	2
	Absolute	....	EC	3
	0 - Page	....	E4	2
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

The specified data is subtracted from the X register. Neither value is affected. The result updates the N, Z and C flags.

COMPARE DATA TO THE Y REGISTER — CPY

FLAGS	Address Mode		Opcode	Bytes
N Z C				
	Implied			
	Immediate	....	C0	2
	Absolute	....	C4	3
	0 - Page	....	CC	2
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

The specified data is subtracted from the Y register. Neither value is affected. The result updates the N, Z and C flags.

DECREMENT — DEC

FLAGS	Address Mode		Opcode	Bytes
N Z				
	Implied			
	Immediate			
	Absolute	....	CE	3
	0 - Page	....	C6	2
	Abs. - X	....	DE	3
	Abs. - Y			
	0 P. - X	....	D6	2
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

The specified data is decremented by 1.

DECREMENT REGISTER X — DEX

FLAGS	Address Mode		Opcode	Bytes
N Z				
	Implied	....	CA	1
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

The X register value is decreased by 1.

DECREMENT THE Y REGISTER — DEY

FLAGS	*Address Mode*		*Opcode*	*Bytes*
N Z				
	Implied	....	88	1
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

The Y register vale is decreased by 1.

EXCLUSIVE-OR ACCUMULATOR-DATA — EOR

FLAGS	*Address Mode*		*Opcode*	*Bytes*
N Z				
	Implied			
	Immediate	....	49	2
	Absolute	....	4D	3
	0 - Page	....	45	2
	Abs. - X	....	5D	3
	Abs. - Y	....	59	3
	0 P. - X	....	55	2
	(Ind.- X)	....	41	2
	(Ind) - Y	....	51	2

DESCRIPTION

The Exclusive OR logic operation is executed between the accumulator and data values. The outcome is placed in the accumulator.

INCREMENT MEMORY BY 1 INC

FLAGS	*Address Mode*		*Opcode*	*Bytes*
N Z				
	Implied			
	Immediate			
	Absolute	....	EE	3
	0 - Page	....	E6	2
	Abs. - X	....	FE	3
	Abs. - Y			
	0 P. - X	....	F6	2
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

The specified data is incremented by 1.

INCREMENT THE X REGISTER INX

FLAGS	*Address Mode*		*Opcode*	*Bytes*
N Z				
	Implied	....	E8	1
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

The value in the X register is incremented by 1.

INCREMENT THE Y REGISTER — INY

FLAGS	*Address Mode*		*Opcode*	*Bytes*
N Z				
	Implied	....	C8	1
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

The value held in the Y register is incremented by 1.

JUMP TO ADDRESS — JMP

FLAGS	*Address Mode*		*Opcode*	*Bytes*
	INDIRECT	....	6C	3
	Immediate			
	Absolute	....	4C	3
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

The address specified is placed in the program counter.

JUMP TO SUBROUTINES — JSR

FLAGS	*Address Mode*		*Opcode*	*Bytes*
	Implied			
	Immediate			
	Absolute	....	20	3
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

The specified address is loaded into the program counter, after pushing the current address + 2 on the stack.

LOAD THE ACCUMULATOR — LDA

FLAGS	*Address Mode*		*Opcode*	*Bytes*
N Z				
	Implied			
	Immediate	....	A9	2
	Absolute	....	AD	3
	0 - Page	....	A5	2
	Abs. - X	....	BD	3
	Abs. - Y	....	B9	3
	0 P. - X	....	B5	2
	(Ind.- X)	....	A1	2
	(Ind) - Y	....	B1	2

DESCRIPTION

The A register is loaded with the data specified.

LOAD THE X REGISTER — LDX

FLAGS	Address Mode		Opcode	Bytes
N Z				
	Implied			
	Immediate	....	A2	2
	Absolute	....	AE	3
	0 - Page	....	A6	2
	Abs. - X			
	Abs. - Y	....	BE	3
	0 P. - Y	....	B6	2
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

The X register is loaded with the data specified.

LOAD THE Y REGISTER — LDY

FLAGS	Address Mode		Opcode	Bytes
N Z				
	Implied			
	Immediate	....	A0	2
	Absolute	....	AC	3
	0 - Page	....	A4	2
	Abs. - X	....	BC	3
	Abs. - Y			
	0 P. - X	....	B4	2
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

The Y register is loaded with the data specified.

LOGICAL SHIFT RIGHT — LSR

FLAGS	*Address Mode*		*Opcode*	*Bytes*
Z C				
	Accumulator	....	4A	1
	Immediate			
	Absolute	....	4E	3
	0 - Page	....	46	2
	Abs. - X	....	5E	3
	Abs. - Y			
	0 P. - X	....	56	2
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

Each data bit is moved one position right. The L.S.B. is placed in the carry, and the M.S.B. is set low.

NO OPERATION — NOP

FLAGS	*Address Mode*		*Opcode*	*Bytes*
	Implied	....	EA	1
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

No operation is performed for 2 cycles.

LOGICAL OR — ORA

FLAGS	Address Mode		Opcode	Bytes
N Z				
	Implied			
	Immediate	....	09	2
	Absolute	....	0D	3
	0 - Page	....	05	2
	Abs. - X	....	1D	3
	Abs. - Y	....	19	3
	0 P. - X	....	15	2
	(Ind.- X)	....	01	2
	(Ind) - Y	....	11	2

DESCRIPTION

The logical OR is performed between the A register and data. The outcome is placed in the A.

PUSH THE ACCUMULATOR — PHA

FLAGS	Address Mode		Opcode	Bytes
	Implied	....	48	1
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

The value in the A register is placed on the stack.

PUSH THE PROCESS STATUS REGISTER — PHP

FLAGS	*Address Mode*		*Opcode*	*Bytes*
	Implied	....	08	1
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

The P.S.R. byte is placed on the stack and the stack pointer updated.

PULL ACCUMULATOR — PLA

FLAGS	*Address Mode*		*Opcode*	*Bytes*
N Z				
	Implied	....	68	1
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

The top byte from the stack is placed in the accumulator and the stack pointer updated.

PULL PROCESS STATUS BYTE FROM THE STACK — PLP

FLAGS	*Address Mode*	*Opcode*	*Bytes*
ALL			
	Implied	28	1
	Immediate		
	Absolute		
	0 - Page		
	Abs. - X		
	Abs. - Y		
	0 P. - X		
	(Ind.- X)		
	(Ind) -Y		

DESCRIPTION

The uppermost byte on the stack is placed in the P.S.R.

ROTATE LEFT — ROL

FLAGS	*Address Mode*	*Opcode*	*Bytes*
N Z C			
	Accumulator	2A	1
	Immediate		
	Absolute	2E	3
	0 - Page	26	2
	Abs. - X	3E	3
	Abs. - Y		
	0 P. - X	36	2
	(Ind.- X)		
	(Ind) - Y		

DESCRIPTION

Each bit of the data is moved one position left. The carry bit is placed in the vacant L.S.B. and the displaced M.S.B. is loaded into the carry.

ROTATE RIGHT — ROR

FLAGS	*Address Mode*		*Opcode*	*Bytes*
N Z C				
	Accumulator	....	6A	1
	Immediate			
	Absolute	....	6E	3
	0 - Page	....	66	2
	Abs. - X	....	7E	3
	Abs. - Y			
	0 P. - X	....	76	2
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

Each data bit is rotated right one position. The carry is placed in the vacant M.S.B. and the displaced L.S.B. is loaded into the carry.

RETURN FROM INTERRUPT — RTI

FLAGS	*Address Mode*		*Opcode*	*Bytes*
ALL				
	Implied	....	40	1
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

Control is returned to the interrupted program. The top byte from the stack is placed in the P.S.R, then the return address is pulled and loaded into the P.C.

RETURN FROM SUBROUTINE RTS

FLAGS	*Address Mode*		*Opcode*	*Bytes*
	Implied	....	60	1
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

The upper two bytes of the stack are pulled. These point to the return address when incremented by 1.

SUBTRACT WITH CARRY

FLAGS N V Z C	*Address Mode*		*Opcode*	*Bytes*
	Implied			
	Immediate	....	E9	2
	Absolute	....	ED	3
	0 - Page	....	E5	2
	Abs. - X	....	FD	3
	Abs. - Y	....	F9	3
	0 P. - X	....	F5	2
	(Ind.- X)	....	E1	2
	(Ind) - Y	....	F1	2

DESCRIPTION

The specified data with borrow is subtracted from the accumulator.

SET THE CARRY FLAG — SEC

FLAGS	Address Mode		Opcode	Bytes
C				
	Implied	....	38	1
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

The carry flag is set.

SET DECIMAL MODE — SED

FLAGS	Address Mode		Opcode	Bytes
D				
	Implied	....	F8	1
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

The decimal flag is set. Data used in the add and subtract operations must be in the binary coded decimal format, until the binary mode is re-entered.

SET THE INTERRUPT FLAG — SEI

FLAGS	*Address Mode*		*Opcode*	*Bytes*
I				
	Implied	....	78	1
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

If set, the I flag prevents either the IRQ or BRK commands from executing.

STORE THE A. IN MEMORY — STA

FLAGS	*Address Mode*		*Opcode*	*Bytes*
	Implied			
	Immediate			
	Absolute	....	8D	3
	0 - Page	....	85	2
	Abs. - X	....	9D	3
	Abs. - Y	....	99	3
	0 P. - X	....	95	2
	(Ind.- X)	....	81	2
	(Ind) - Y	....	91	2

DESCRIPTION

The specified location is loaded with the value in A.

STORE THE X REGISTER — STX

FLAGS	*Address Mode*		*Opcode*	*Bytes*
	Implied			
	Immediate			
	Absolute	....	8E	3
	0 - Page	....	86	2
	Abs. - X			
	Abs. - Y			
	0 P. - Y	....	96	2
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

The specified location is loaded with the value in the X register.

STORE THE Y REGISTER — STY

FLAGS	*Address Mode*		*Opcode*	*Bytes*
	Implied			
	Immediate			
	Absolute	....	8C	3
	0 - Page	....	84	2
	Abs. - X			
	Abs. - Y			
	0 P. - X	....	94	2
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

Place the value held in the Y register in the location specified.

TRANSFER ACCUMULATOR TO THE X REGISTER — TAX

FLAGS	*Address Mode*		*Opcode*	*Bytes*
N Z				
	Implied	....	AA	1
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

The value held in the accumulator is placed in the X register. The accumulator data is unchanged.

TRANSFER THE ACCUMULATOR TO THE Y REGISTER — TAY

FLAGS	*Address Mode*		*Opcode*	*Bytes*
Z N				
	Implied	....	A8	1
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

The contents of the accumulator are placed in the Y register.
The A register data is unchanged.

TRANSFER THE STACK POINTER TO X — TSX

FLAGS	*Address Mode*		*Opcode*	*Bytes*
Z N				
	Implied	....	BA	1
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

The Stack pointer is placed in the X register.

TRANSFER A. TO X — TXA

FLAGS	*Address Mode*	*Opcode*	*Bytes*	
Z N				
	Implied	....	8A	1
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

The value held in the X register is placed in the A register.

TRANSFER X TO S — TXS

FLAGS	*Address Mode*		*Opcode*	*Bytes*
	Implied	....	9A	1
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

The value held in the X register is placed in the Stack pointer register.

TRANSFER Y TO A — TYA

FLAGS	*Address Mode*		*Opcode*	*Bytes*
Z N				
	Implied	....	98	1
	Immediate			
	Absolute			
	0 - Page			
	Abs. - X			
	Abs. - Y			
	0 P. - X			
	(Ind.- X)			
	(Ind) - Y			

DESCRIPTION

The value held in the Y register is placed in the A register.

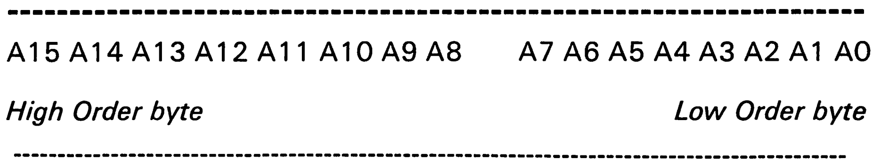
CHAPTER 4

ADDRESSING TECHNIQUES

Literals and Absolutes. Zero page addressing. The Indirect approach. Binary Coded Decimal. Floating points.

Literals & Absolutes

Any memory location from 0-65535 may be specified by 2 bytes. The most significant eight bits are referred to as the High Order byte, and the least significant eight, the Low Order byte. These are A8 to A15, and A0 to A7 in the following diagram:



A data location given in this way is one of the ten formats that the 6510 will accept with an instruction.

It must be stressed that the reason each type of instruction may have more than one opcode is to permit address format specificity.

For example, the LDA category contains eight opcodes. The same operation is performed by each, but the data address is interpreted in a different way.

The reader will find the majority of formats, listed in table 4.1, to be straightforward.

IMPLIED

No address is necessary - it is implicit to the instruction.
Example: DEX - decrement the X register.

IMMEDIATE

The byte following the opcode contains the actual data value, not an address.

FORMAT	*BYTES*	*DESCRIPTION*
Implied	0	Implicit
Immediate	1	Actual data - No address
Absolute	2	16 bit address, reversed
Zero page	1	
Zero page - X	1	X register + byte value
Zero page - Y	1	Y register + byte value
Absolute X	2	X register + bytes value
Absolute Y	2	Y register + bytes value
(Indirect,X)	1	Pointer @ (X + byte)
(Indirect),Y	1	Pointer + Y register

Table 4.1 The 6510 Address formats

Example: LDA= &10 - Load the accumulator with 16 decimal.
(The = sign will be used to indicate this format).

ABSOLUTE

The address is given by the two bytes following the instruction. On the 6510, the High and Low order bytes are reversed.

Example: STA &00 &FF - Store the value held in the accumulator in location & FF00 (65280 decimal).

ZERO PAGE

The data lies in zero page at the address given by the byte following the instruction.

Example: ASL &20 - The Arithmetic Shift Left is performed on location &20.

ZERO PAGE-X ZERO PAGE-Y

The zero page data address is the sum of the specified index register and the byte with the instruction.

Example: CMP &30 - Data location = &30 + Contents of the X register.

ABSOLUTE X

The target location is the sum of the contents of the X register and the 16 bit address following the instruction. Again, in the long address, the bytes are reversed.

Example: SBC &00 &FF - Subtract the byte at location (& FF00 + contents of the X register) from that held in the accumulator.

ABSOLUTE Y

As for ABSOLUTE X, albeit the Y register is used.

(INDIRECT,X)

The final address is held by two bytes in zero page. The lower of these is pointed to by (value in X register + the short address byte). The low order byte of the absolute address occurs first.

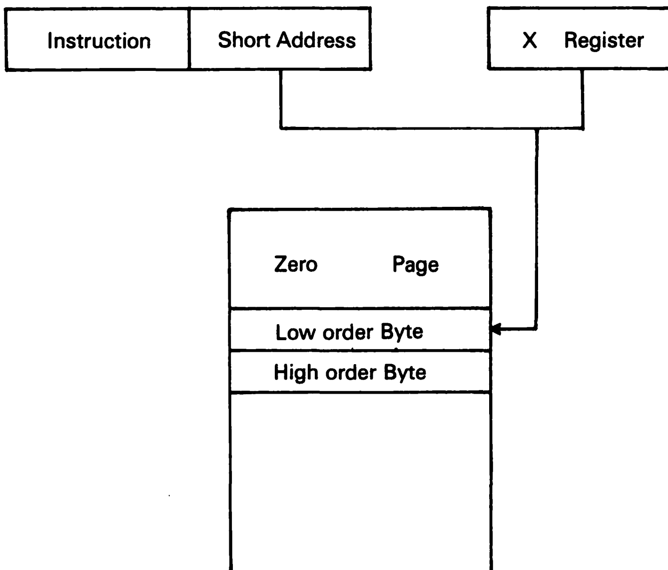


Figure 4.1 The Indirect, X address format

Example: ORA &14 - The Logical OR is performed between the address given by the two zero page bytes and the accumulator.

(INDIRECT),Y

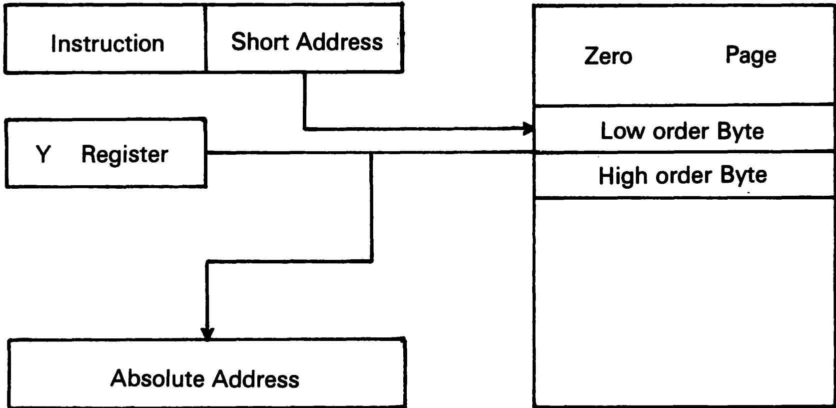


Figure 4.2 The Indirect, Y address format

The short address with the instruction points to the low order byte of an absolute address in zero page. This is added to the value held in the Y register.

Example: LDA &04 - Load the accumulator with the absolute address equal to the Y register plus the value held in locations 4 and 5.

Both forms of indirect addressing are indexed by a register.

However, the jump instruction may use a simpler form in which the instruction is accompanied by an absolute address which points to the Low Order byte of the final address. JMP is the only instruction that may use the 'pure' indirect format.

Example: JMP &00 &FF - Control is directed to the absolute address held in locations &FF00 and &FF01.

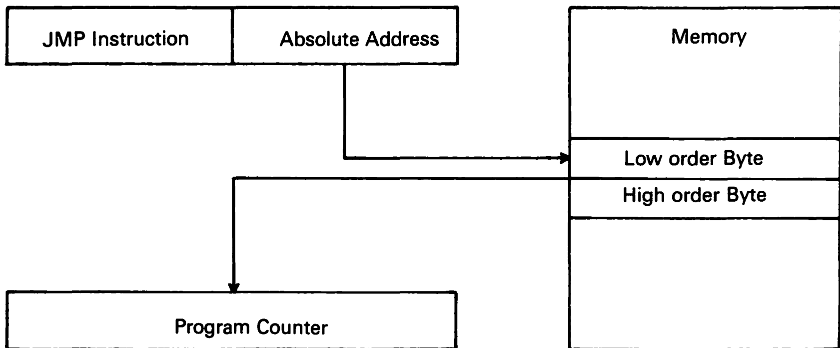


Figure 4.3 The Indirect address format

Pro's and Con's

Although the indexed and indirect address formats allow blocks of data to be manipulated with relative ease, their execution time is relatively slow. An opcode using the immediate form of address will usually require 2-4 cycles less than the indirect format.

The (Indirect, X) format is widely used to select one from a table of vectors in zero page. Here, the short instruction byte points to the table's base address with vector selection by the X register.

A principle use of the (Indirect),Y mode is in operations on sequential locations. For instance, to decrement the 200 values from &C000 to &C0C8, the base address, &C000 is first placed in zero page. After each decrement, the Y register is incremented using the INY operation. The program is listed below.

Address	Hex Code	Mnemonic	Comment
679	A0	LDY	Load the Y register
680	0		with 0
681	A9	LDA	Load the Accumulator
682	0		with 0
683	85	STA	Store A. in zero page
684	FB		location 251
685	A9	LDA	Load the Accumulator
686	C0		with 192
687	85	STA	Store A. in zero page
688	FC		location 252
689	B1	LDA	Load A. indirectly
690	FB		with (FC+FB)+Y
691	AA	TAX	Transfer A. to X
692	CA	DEX	Decrement X
693	8A	TXA	Transfer X to A.
694	91	STA	Store A. in location
695	FB		(FC+FB)+Y
696	C0	CPY	Compare Y
697	C8		with 200
698	D0	BNE	Branch if not = 0
699	01		to PC + 1
700	60	RTS	Return from sub.
701	C8	INY	Increment Y
702	4C	JMP	Jump to location
703	B1		689
704	02		

Binary Coded Decimal

It may be felt that machine code is quite difficult enough without the hardships of decimal-hexadecimal conversion, a sentiment that perhaps led the designers of the 6510 to incorporate the facility to allow the processor to use the binary coded decimal notation in certain operations.

In chapter 1, we mentioned that each byte can be viewed as two nibbles, each holding a value from 0-15. In binary coded decimal, a nibble is used to represent one digit of a decimal number - thus a byte may represent a value in the range 0-99.

Although B.C.D. imposes a severe limitation on range in comparison to binary, the additional convenience it offers is not inconsiderable.

The B.C.D. mode is entered using the SED (SEt Decimal mode) instruction, opcode &FB, which also sets the decimal flag in the P.S.R. To re-enter the binary mode use the CLD (CLear Decimal mode) instruction, opcode &D8.

Decimal	*Binary coded decimal*
0	0 0 0 0 0 0 0 0
8	0 0 0 0 1 0 0 0
10	0 0 0 1 0 0 0 0
20	0 0 1 0 0 0 0 0
99	1 0 0 1 1 0 0 1

Figure 4.4 B.C.D. Representation

Floating point

If the 64 is manipulating integer variables, which take values in the range -32768 to 32767, it uses two bytes to store the value. When floating point variables are in use, 5 bytes are required to accomodate the increase in range. Further, the value is converted into two sections, an exponent and mantissa. The exponent is the power of two by which the mantissa is multiplied to obtain the original value. The exponent requires one byte, leaving the mantissa the four low order bytes.

As the mantissa is the fractional part, it is helpful to be familiar with how this is represented.

Decimal fraction	*Binary*
0.5	0.1
0.25	0.01
0.125	0.001
0.0625	0.0001
0.03125	0.00001

Figure 4.5 Binary fraction equivalents

The division by two of a binary value is achieved by moving each bit one position right. Therefore, to obtain the binary equivalent of 1/2, the byte 00000001 is shifted to give 0.1 Figure 4.5 shows the column values after the binary point.

When performing operations on floating point variables the BASIC interpreter routines make use of two floating point accumulators located in the zero page. These have no relation to the processor's accumulator, each consisting of five locations for the storage of a f.p. variable (the exponent has &80 added). An adjacent byte holds the sign.

As we stated earlier, accumulator 1 is also used by the USR function to pass a variable between BASIC and code routines.

The resident routines that perform mathematical operations are easily accessed, and unlike those of certain machines, do not consist primarily of defects! So unless you require a greater degree of accuracy than these provide, there is little point in writing your own.

Multiple Precision

As we have seen, if the result of an addition exceeds 255 the carry flag is set. So when performing the addition of multi-byte values it is necessary only to clear the Carry flag for the initial addition. On completing the final addition, a common programming mistake is to omit the check for a final carry.

The Overflow flag

We have already seen that subtraction is performed by the processor using the Two's Complement notation. To do this, the byte to be taken from that in the accumulator is actually added to it - after conversion into the two's complement form. The conversion is achieved by inverting each digit and adding the carry bit. It should now be clear why the carry is set prior to a single byte subtraction.

In two's complement notation, a byte holding a negative value will always have the M.S.B. set. If during an operation, that bit is accidentally altered, the overflow flag is set.

This will occur when:

1. There is no external carry, but one from bit 7 to 8
2. There is an external carry, but no carry from bit 7 to 8

The stage has now been reached when the readers may wish to try a few of their own routines. The following guidelines may be helpful:

i. Always save a machine code program before a trial run. If it contains a bug the 64 will probably hang up and need to be switched off before it may be used again.
ii. A code routine is unlikely to operate correctly straight away, if you are very lucky one or two sections may perform as expected. Patience is the keyword.
iii. If combining a M/C routine with a BASIC program, it is easier to use a BASIC loop to poke the code into position, rather than loading and saving the two programs separately.
If values are to be passed between the two, unless the resident f.p. routines are being utilized, use peeks and pokes in preference to the USR function.
iv. Generally, M/C routines require more debugging than their BASIC counterparts, and it is sensible to structure the program into small sections each of which is called by a core program.

CHAPTER 5

STACKS & SUBROUTINES

More on Interrupts. Stack instructions. Subroutine Use. Program structure. Positional Independence.

More on Interrupts

In addition to the execution of user routines the processor has to service a variety of system functions. The two are dove-tailed, using interrupts. We briefly introduced these in chapter 2. Essentially, program execution is temporarily diverted to a pre-set location until an RTI is encountered. The 6510 supports 3 types of interrupt:

1. Non Maskable Interrupt (NMI)
2. Break (BRK)
3. Interrupt Request (IRQ)

The Non Maskable interrupt is rarely used and is of little interest to the programmer. BRK and IRQ are identical in function but are initiated by different means. BRK is a software interrupt and is forced using opcode &00, while the IRQ is initiated by bringing low the 6510's IRQ pin.

Either interrupt will set the P.S.R. I flag. If the request is made with the flag clear, the interrupt is accepted, otherwise it is negated.
If enabled, the sequence is as follows:
1. The 6510 completes the current instruction.
2. The P.C. & P.S.R. are saved to the stack.
3. A jump is made to the address in locations &FFFE and &FFFF

Hex location	*Interrupt*		
FFFD	Reset	:	Address high order byte
FFFC			low order byte
FFFB	NMI	:	Address high order byte
FFFA			low order byte
FFFF	IRQ	:	Address high order byte
FFFE			low order byte
FFFF	BRK	:	Address high order byte
FFFE			low order byte

Figure 5.1 The Interrupt vectors

As both interrupts are serviced by the same routine its first function is to identify the interrupt responsible. If an IRQ was used, the service routine branches to the address pointed to by the vector at locations 788-9. The two subsequent locations hold the BRK vector.

If the reader wishes to replace either procedure, it is simply a matter of altering the vector. However, as the IRQ routine is used to scan the keyboard and update TI, care should be exercised if these functions need to be maintained.

Stack instructions

The stack was briefly introduced in chapter 2. As we have seen it has three principal uses:

i. To hold subroutine return addresses
ii. To preserve the P.S.R. and provide the return address during interrupts
iii. To act as a temporary storage area for program data

Either the accumulator or the P.S.R. registers may store data to or from the stack. Although the contents of the program counter are placed on the stack during subroutines and interrupts, no instruction is available to the programmer for this.

Placing a value on the stack is termed 'pushing' the stack. The variants of this are PHA (PusH Accumulator) and PHP (PusH P.S.R.). The reverse operation is known as 'pulling' or 'popping' the stack. This is accomplished using the PLA or PLP instructions.

The 'height' of the stack is held by the stack pointer register, which gives the address of the next free byte. It may be amended by two transfer instructions. These either load the value in the X register into the S.P. (TXS) or load the X register with the S.P. value (TSX).

The operation of the stack is automatic and it is not necessary to manually update the S.P.

Subroutine Use

The function of a subroutine is threefold:

1. To avoid unecessary duplication
2. Simplify program design
3. Afford positional flexibility

In Commodore BASIC when several subroutines are nested it is not possible to return to the main program and omit returns to intervening routines. In M/C where multiple layers of subroutines are not infrequent, this is easily achieved by pulling the return addresses from the stack.

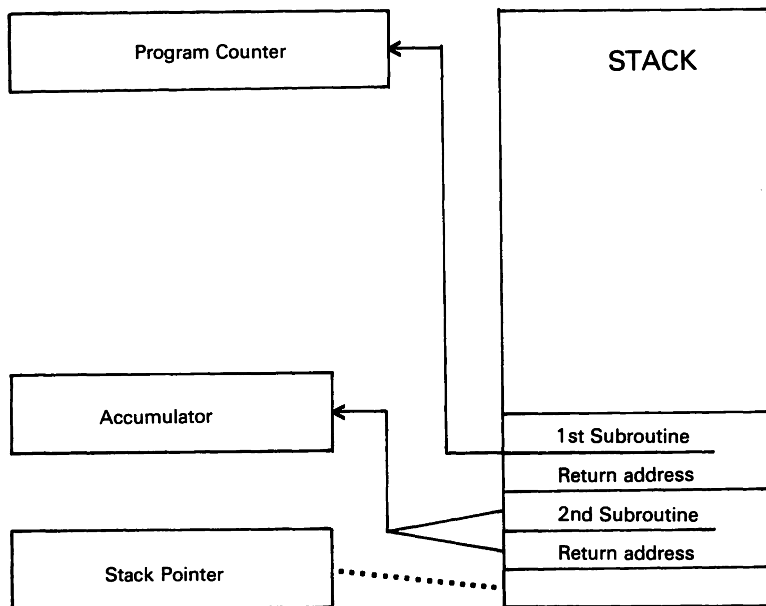


Figure 5.2 Subroutine return negation

Two main disadvantages are associated with subroutine use. As machine code routines often require repetition to the order of thousands of times, the execution of two control instructions per call may incur an unacceptable time penalty.

Secondly, prior to the call, the general registers often need to be preserved on the stack. Again, this incurs a time penalty, but in addition, stack space, which may be at a premium, is used.

Program structure

We mentioned, briefly, the need to structure a machine code program. Beyond a division into stages, this will entail decisions about the priority of any section to the limited zero page space. For instance, block operations may require a two byte pointer whereas a queue structure, similar to the stack, will only need a single byte.

In the standard system configuration, there are only 5 free zero page locations: 2 and 251-4.

In general; if the final form of the program cannot be predicted at the outset, or space is to be allotted for expansion, it is usual to insert blocks of NOP's.

Sorting

Prior to the analysis of a set of data, the elements must be placed in some kind of order. Whether or not this is on a numeric or alphabetic basis, each element will ideally consist of a label (typically 3 bytes) followed by a data portion of fixed length. This allows an element to be inserted or deleted by moving the remainder of the table the relevant number of positions.

A target element is located, either by a search from the first element forwards (Serial search) or from the mid-position, gravitating towards the desired element (Binary search).
The two procedures to be incorporated in each check are:

i. For end of table run off
ii. The search is not for a non-existent element

It is common to begin sampling with large step intervals. As the target approaches these are reduced.

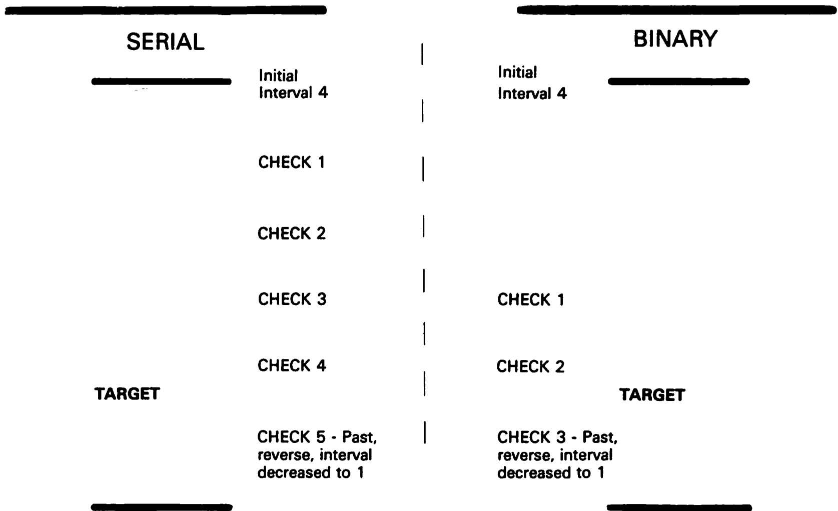


Figure 5.3 Binary and Serial search procedures

Positional Independence

A final factor to consider at the design stage is whether the program, or section(s) of it, need to be written in such a way that their operation is not affected if re-positioned. For example if the program is a utility, then positional independence is often obligatory.

CHAPTER 6

SCREEN AND KEYBOARD TECHNIQUES

The C.I.A. 1. The Interrupt routines. SCNKEY & Associates. Function Keys. Sprites & Animation.

The C.I.A.1

Each key on the keyboard generates a specific code. After identification, the code is temporarily stored in location 197.

The first stage of this process is performed by the C.I.A. 1 which has a line to each of the eight rows and columns of the keyboard. When a key is depressed the number of the row is registered in location 56321 and the column in location 56320.

The process is completed by the operating system. Each 1/60 of a second, a timer on C.I.A.1 initiates an IRQ. The interrupt handling routine then checks to see if a character is to be registered, and if so, it identifies the code and updates several system locations.

After key identification, the next stage centres on the keyboard queue from locations 631 to 640. As each keypress is registered, the character code (as distinct from the key code) is placed in the next free byte of the queue, and the character count, held in location 198, is incremented.

When a carriage return (ASCII code 13) is encountered, the input to that point is executed.

By using a short M/C routine to place a BASIC command with carriage return in the keyboard queue, the BASIC interpreter can be made to execute instructions without keyboard entry. However, the queue is only processed when the character count in location 198 is more than zero.

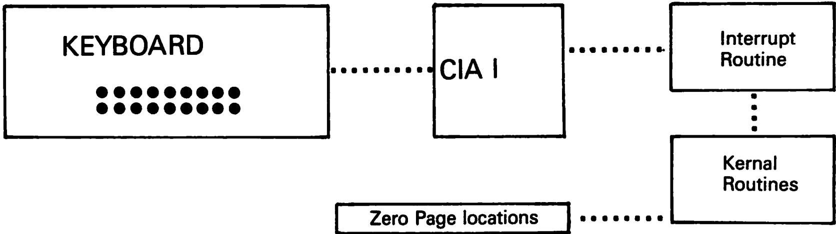


Figure 6.1 Keyboard input route

The procedure is illustrated by the short program listed below. The line: 1 LO. (RETURN) is placed in the queue followed by the abbreviated form of RUN (RETURN). The abbreviations result from the addition of 128 to the ASC II code of the second keyword character.

Location	*Hex*	*Mnemonic*	*Comment*
679	A9	LDA	Load A. with
680	31		49
681	8D	STA	Store A. in the first
682	77		buffer location
683	02		
684	A9	LDA	Load A. with
685	4C		76
686	8D	STA	Store A. in the next
687	78		buffer location
688	02		
689	A9	LDA	Load A. with
690	CF		207

691	8D	STA	Store A. in the next
692	79		buffer location
693	02		
694	A9	LDA	Load A. with
695	0D		13
696	8D	STA	Store A. in the next
697	7A		buffer location
698	02		
699	A9	LDA	Load A. with
700	52		82
701	8D	STA	Store A. in the next
702	7B		buffer location
703	02		
704	A9	LDA	Load A. with
705	D5		213
706	8D	STA	Store A. in the next
707	7C		buffer location
708	02		
709	A9	LDA	Load A. with
710	0D		13
711	8D	STA	Store A. in the next
712	7D		buffer location
713	02		
714	A9	LDA	Load A. with the No.
715	07		queue of characters
716	85	STA	Store A. in location
717	C6		198
718	60	RTS	Return from subroutine

The treatment of registered characters will depend on whether the 64 is operating in direct or program mode. In direct mode, each character is placed in the keyboard buffer and on the screen.

As each is processed, the remaining characters are shifted one position forward and the character count is decremented. In the program mode, input from the keyboard is only processed via the INPUT and GET statements.

During I/O operations the keyboard scan is limited to the RUN/STOP and RESTORE keys which are serviced by separate routines.

The Interrupt routines

When an interrupt occurs, the service routine identifies the source and branches to the relevant routine. The C.I.A. and V.I.C. chips may initiate interrupts for several reasons, either to service an external device or an internal function.

I/O interrupt control is facilitated by the interrupt control register on each C.I.A.. The C.I.A. 1 chip can only initiate IRQ's, and the C.I.A. 2 only NMI's.

Each control register is best thought of as two registers - the mask register, which may only be written to and the data register, which may only be read.

Each has five possible sources of interrupt. These are flagged by the five low order bits in each register. When the source wants to initiate an interrupt, it sets the allotted bit in the data register. If the interrupt is 'enabled' by the corresponding bit in the mask register being set, the IRQ or NMI to the 6510 occurs.

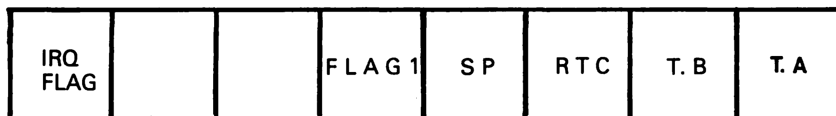
Thus interrupts from a particular source may be blocked by clearing the appropriate bit in the mask register. However, it is not a simple case of writing the required configuration to the register.

The M.S.B. of the byte written determines how the write is interpreted. If set, the mask register bits will be set in those positions the data has bits set. The other locations remain unchanged. Conversly, if the M.S.B. is set low, all the data bits set high cause the equivalent bits in the mask register to be set low and leave the others unchanged.

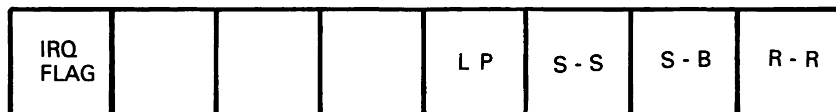
An enabled IRQ or NMI will also set the M.S.B. in the data register. A read of the register will clear it. As this is necessary in order to identify the interrupt source, the reset is not inefficient.

Figure 6.2 shows the two IRQ control registers. The FLAG 1 bit is used by either the Serial bus SRQIN line or cassette read line. The SP bit has a line to the User port, and the C.I.A.'s timers (each C.I.A. has two timers and a real time clock) utilise the two low order bits. An alarm facility is available from the real time clock, and this uses the third bit.

The V.I.C. 2 interrupts are handled in a slightly different manner. Two registers are involved: the mask register at location 53274 and the flag register at 53273. Their functions mirror the mask and data registers on each C.I.A. As there are only four sources of interrupt, only the low order nibble is used to flag the requests. Again, Figure 6.2 shows the format.



56333 C.I.A. 1



53273 V.I.C. 2

Figure 6.2 The IRQ data registers

The LP bit is used by the light pen, S-S and S-B flag sprite / sprite and sprite / background collisons and the l.s.b. indicates a match between the raster count and the value held by the raster registers (see appendix G).

Unlike the C.I.A. interrupt registers, a read of the V.I.C. flag register will not affect the contents. As the M.S.B. is set on the occurrence of an enabled interrupt, both it and the source bit, need to be cleared.

For either the C.I.A. or V.I.C. interrupts the technique of polling may be used to provide a second set of lower priority 'interrupt' events. Here the mask register bit is not set, instead the data byte is polled or checked to see if a request has been received.

Before leaving this area, it is worth mentioning that where access to the character R.O.M. is necessary, the normal interrupt activity must be suspended. If working in code, use the SEI instruction, otherwise amend the C.I.A.1 control register A by setting the L.S.B. low. This stops timer A & is achieved by ANDing the register with 254. To restart the timer, the location is OR'd with 1.

SCNKEY and Associates

The operating system contains a table of vectors to various resident routines. The purpose of this table or Kernal is twofold.

First, it allows the programmer simple access to the resident routines. Second, it provides Commodore with a method of updating the routines and their addresses without affecting existing software.

Three kernal routines are directly concerned with keyboard functions. Before discussing them it is appropriate to give some general guidelines regarding the use of the vector table:

1. Prior to the call, save all necessary data in the general registers to the stack.
2. Do not use the routines unless there is a reasonable margin of storage capacity available in the stack.
3. Preliminary data may be required. This is passed to the routine via the general registers.
4. Other initialization routines may need to be called before to the target routine.

Keyboard handling

If a key is pressed, the SCNKEY routine is responsible for placing the appropriate code in the keyboard queue and updating the character count in location 198.

Next, the GETIN routine is called to transfer the first code in the queue to the accumulator. Any remaining codes are moved up a position and the character count decremented by 1. If the queue is empty the accumulator is loaded with zero. The SCNKEY vector is at &FF9F and the GETIN vector at &FFE4.

A sibling of these is CHRIN. This takes a byte from the open input channel (Keyboard default) and places it in the accumulator. If the channel is the keyboard, then the byte is taken from the BASIC input buffer. The buffer will accommodate upto 88 bytes. The CHRIN vector is at &FFCF.

In addition to calling SCNKEY, the interrupt handling routine updates the system clock by a call to the UDTIM routine, vector: &FFEA. This procedure is a required preface to the STOP routine, which will set the Z flag if the STOP key is pressed during the UDTIM call. Clearly, if the interrupt handling routine is dispensed with, the UDTIM and STOP routines must be called for the STOP key to remain functional.

Simple keyboard handling may be achieved by loading the accumulator with the contents of location 197 (A value of 64 is returned if no key is depressed) and comparing it with the action options. Location 653 is the flag byte for the shift/Commodore/Control keys.

During execution of a BASIC program, each line, in tokenized form, is placed in the input buffer. The CHRGET routine, at locations 115-138, subsequently places each byte into the accumulator for the interpreter to operate on. The buffer extends from locations 512-600. A pointer at &7A and &7B holds the address of CHRGET's target byte. After checking that th byte does not hold the code for a space, 32, the routine sets the carry flag on the basis of the ASC II code.

To add or change BASIC commands, a common method used is to insert a user wedge into the CHRGET routine. Typically, this will redirect control to a subroutine to check for the change. If this is not found, control is handed back to CHRGET.

The Function keys

In general these are best used by inserting a wedge into the interrupt handling routine. The key may be identified by examining location 197 in conjunction with location 653.

Figure 6.4 lists the combinations generated.

In order to do this, control is re-directed to the users subroutine by changing the interrupt handling vector at &0314-5.

Function Key	*Value in 197*	*Value in 653*
F1	4	0
2	4	1
3	5	0
4	5	1
5	6	0
6	6	1
7	3	0
8	3	1

Figure 6.4 Function Key codes

As an interrupt may occur during replacement of the vector, set the interrupt flag prior to the reset - and clear it afterwards!

At the end of the routine the handler is re-engaged by a jump to &EA31. In practice, once the key is identified, the most straightforward way of

implementing the required functions is to place character codes in the keyboard queue.

On entry to the interrupt routine, the general registers may contin data, so the first action of the wedge should be to push them on the stack.

Sprites and Animation

The use of sprites in machine code is similar to their use in BASIC. The main difference is the delay necessary between moves. The most common method of achieving this is to -

1. Zero the X and Y registers
2. Increment one until it returns to zero
3. Increment the other.

The process is repeated until the second register returns to zero. Obviously, to vary the delay, any increment that divides into 256 may be used. Try two as a starting point. Increasing the loop increment may give a simple method of altering the skill level in an arcade type game.

Sprite collisions are handled by examining the two flag bits in the V.I.C. 2 interrupt data register. These will remain set after the first collision.

The simplest method of analysis is to AND the register with a byte with only the target bit set. If the result is zero, no collision occurred.

The ability to selectively enable the interrupts permits excellent flexibility, and warrants the time required to prepare the interrupt wedge.

Using sprites to achieve good animation is simply a matter of observing a few ground rules :

1. Jerky movement is avoided by selecting a small step size
2. Following the correct sequence -
    i. Place object on screen for time X
    ii. Delete object for time X
    iii. Place it at the new location

Writing and Using Larger Programs

We have now reached the stage where you will need an alternative to manually poking programs into memory.

The main options are:

1. *A hex-loader:* Most machine code listings are given in hexadecimal, and to type them in, converting to decimal as you go, is a little laborious. One way round this is to write a short BASIC program that accepts two digit hexadecimal numbers and pokes them into memory.

The program below is suitable for most listings but if you intend to spend more than a few hours at the keyboard an assembler is an essential.

```
**************
* Hex Loader *
*            *
**************

10 POKE 53280,0:POKE 53281,0:PRINT CHR$(147):PRINT
20 INPUT"START LOCATION";S:AA=S
30 PRINT CHR$(147):PRINT:PRINT"         HEX    DECIMAL    Q TO
FINISH"
40 PRINT S;"  ";
50 GET A$:IF A$="" THEN 50
52 IF A$="Q" THEN 400
60 IF ASC(A$)>47 AND ASC(A$)<58 THEN 72
70 IF ASC(A$)<65 OR ASC(A$)>70 THEN 50
72 PRINT A$
80 GET B$:IF B$="" THEN 80
82 IF ASC(B$)>47 AND ASC(B$)<58 THEN 100
90 IF ASC(B$)<65 OR ASC(B$)>70 THEN 80
100 PRINT B$;"    ";
110 IF VAL(A$)<>0 OR A$="0" THEN 130
120 A$=MID$("101112131415",1+(ASC(A$)-65)*2,2)
130 A=VAL(A$)
140 IF VAL(B$)<>0 OR B$="0" THEN 160
150 B$=MID$("101112131415",1+(ASC(B$)-65)*2,2)
160 B=VAL(B$):P=16*A+B:POKE S,P:PRINT P:PRINT
200 S=S+1:GOTO 40
400 FOR X=1TO8:PRINT CHR$(157);:NEXT:PRINT " START :"AA;" FI
:"S-1
```

2. *An assembler:* The majority of commercial packages consist of several sections:

A. Assembler/Editor
B. Monitor
C. Disassembler

The assembler is used to write a program using mnemonics in place of opcodes and if required, strings instead of addresses. This simplifies the process considerably and avoids the manual calculation of such awkward things as branch displacements.

Once the 'source file' is complete it is assembled into code and placed in memory. The user then saves the source and enters the monitor to test run the program - a step at a time if necessary. The disassembler is used to examine and alter areas of memory and is usually accessed from the monitor.

In addition to these facilities, a set of secondary operations will be supported. Typically these will include moving/saving/loading blocks of memory and source files, search operations for strings in various formats etc.

As the 'assembler' provides the programming environment, choosing the right unit is important.

The price range is considerable - from a few pounds upwards. Several of the longer programs featured in this book were written using the ZEUS 64 ASSEMBLER - a cassette based package. This low cost system from Crystal Computing features a reasonable range of facilities (including single letter commands) and is priced at about £10.

Going upmarket the choice is between disc and cartridge. A possible disadvantage with a cartridge is that one storage medium will be assumed which can be a little tedious if you use the other. On the other hand the time and fuss of loading the system in is avoided. The Commodore disc based Editor/ Assembler is priced at around £24.

Is it worth paying more for the extra facilities? Not really, unless you envisage spending a great deal of time at the keyboard or want to avoid waiting for the system to load up each time you use it.

However, if you have a friendly bank manager the most important features to look for are:

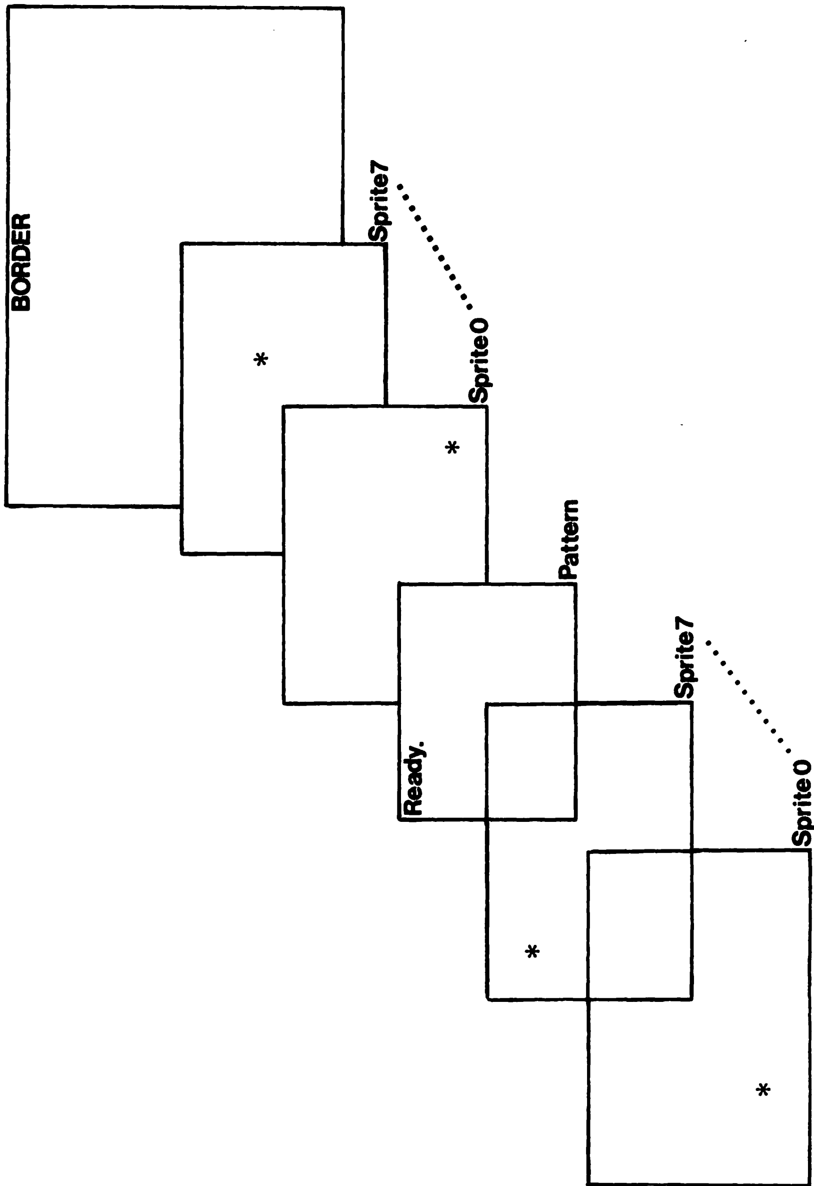


Figure 6.5 Display Planes

1. *Linker*
2. *Macros*
3. *Ability to place code in BASIC programs*

A linker allows you to assemble and test sections of the source file independently. Quite a useful facility but only necessary for the development of large programs.

The ability to insert a pre-defined section (macro) of the source file at any location can save a lot of repetitive typing - but again only useful for major projects.

In the program listings that follow, both the source file and hex dumps are given. Up to this point we have indicated a hexadecimal number with the ampersand (&) sign and the immediate form of address with the equate (=) sign.

Alternatives to these in common use are the dollar sign ($) for the hexadecimal form and the hash symbol (#) for the immediate address format.

Graphics and the 64

Although the reader is assumed to have a familiarity with the graphics facilities on the 64, their use with machine language permits many more ambitious effects to be realised especially using a bit mapped screen. The remainder of this chapter gives a quick overview of the graphics facilities and ends with several detailed examples.

Essentially the 64 will display either a text or bit mapped screen, or by the use of raster interrupts a mixture of both may be obtained. Sprites are available in both modes up to a maximum of 8 on screen with up to four on one screen line. Sprites have a fixed priority from 0 - 7 with sprite zero overlaying any other.

Additionally, the priority of each sprite to the background may be set. Sprites are controlled using the sprite to sprite and sprite to background collision registers in conjunction with the capability to pull interrupts on sprite - sprite and sprite - background collisions.

Independent of the display mode selected, the VIC 2 chip derives its output using information scanned from a 16K block of memory (or which appears to it to reside in that block).

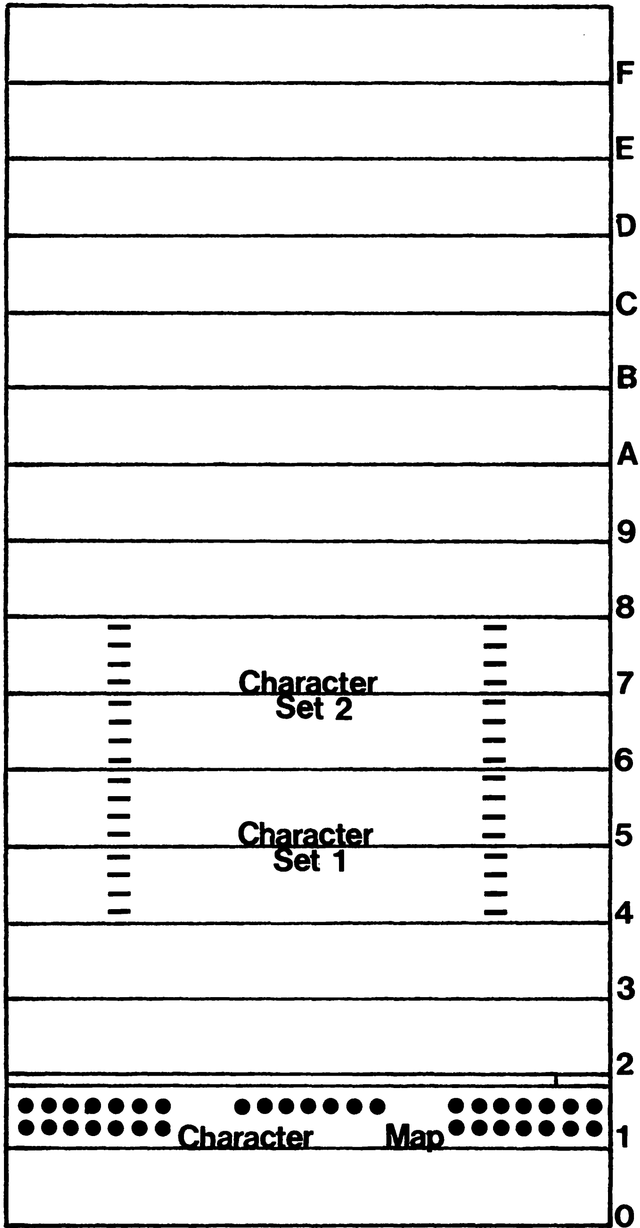


Figure 6.6 Default Display Configuration

The block selected may be 0-16K, 16-32K, 32-48K or 48-64K. On power up the block from 0-16K is used to produce a text screen.

The bank is specified by placing a value of 0-3 in the two low order bits of Port A of CIA #2 - after they have been set to outputs by setting the two least significant bits of the data direction register. The port appears at location 56576 and the direction register at 56578.

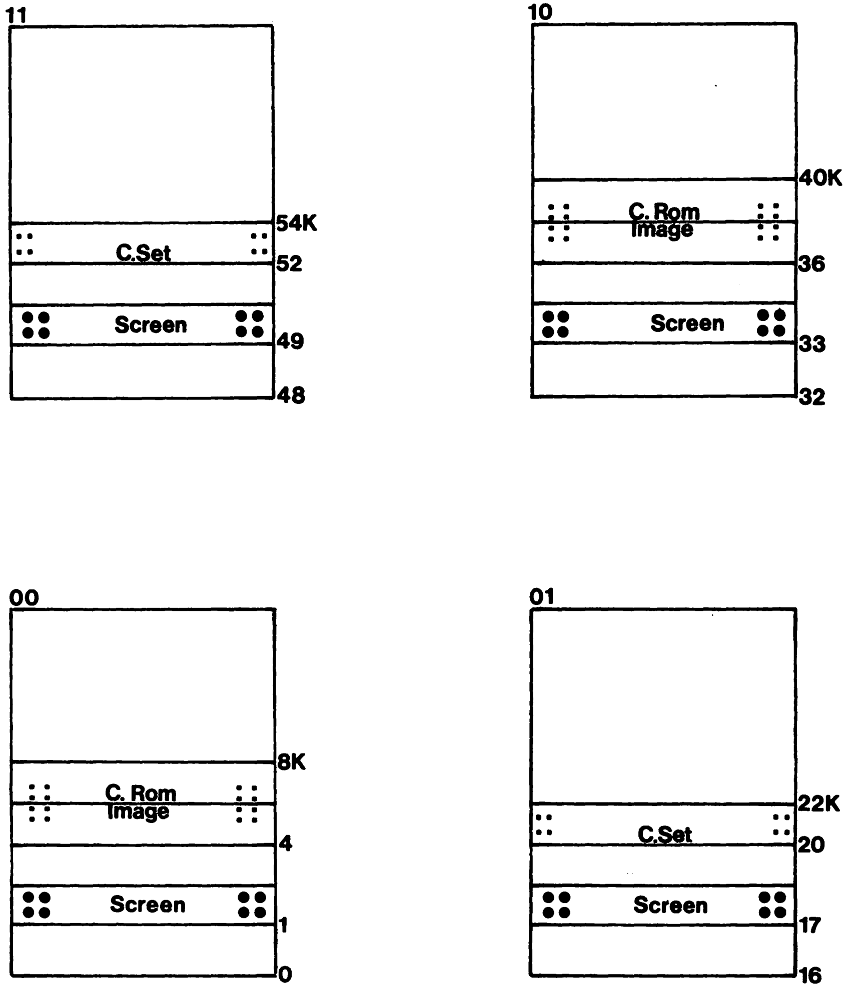
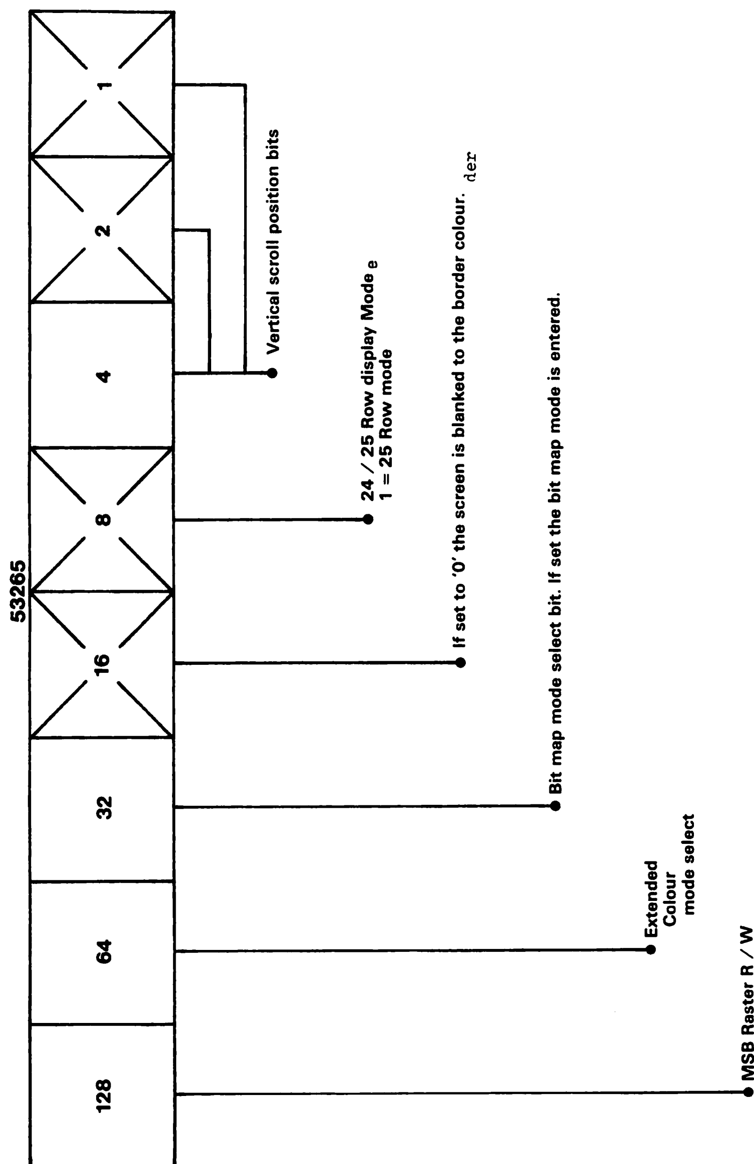


Figure 6.7 The four possible VIC 2 'banks' of memory. Selection is by bits 0 and 1 of location 56576 given top left.

The three principal registers involved with display manipulation appear at locations 53270, 532372 and 53265.



Figire 6.8 VIC 2 Register 17 Location 53265

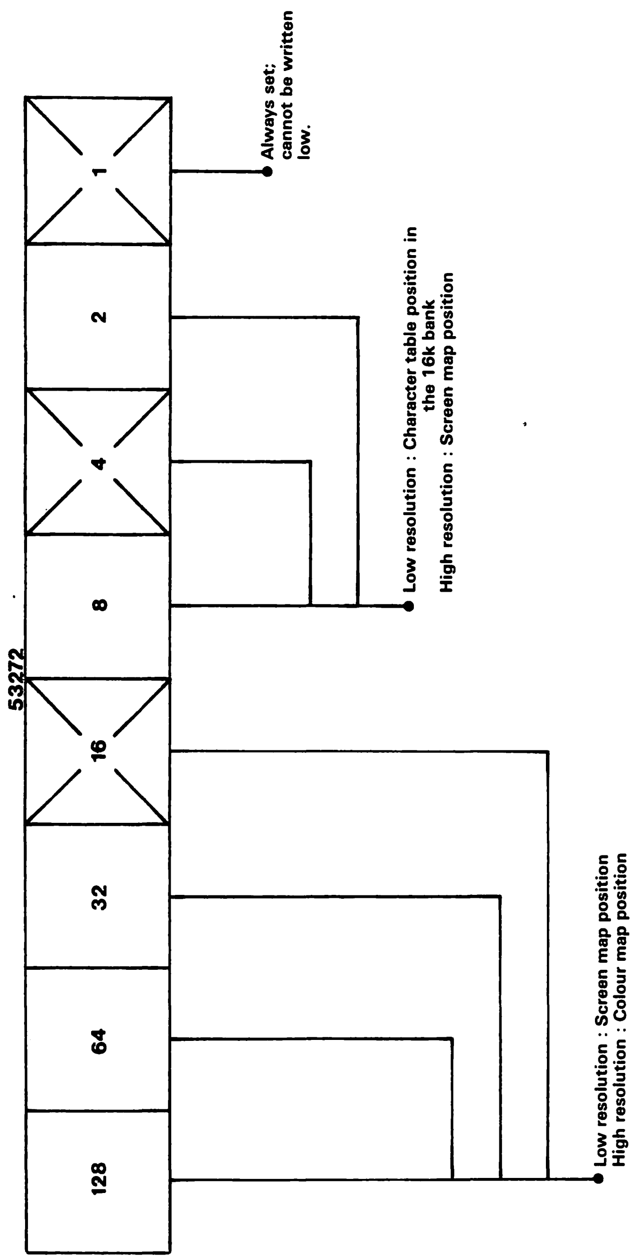


Figure 6.9 VIC 2 Register 24 Location 53272

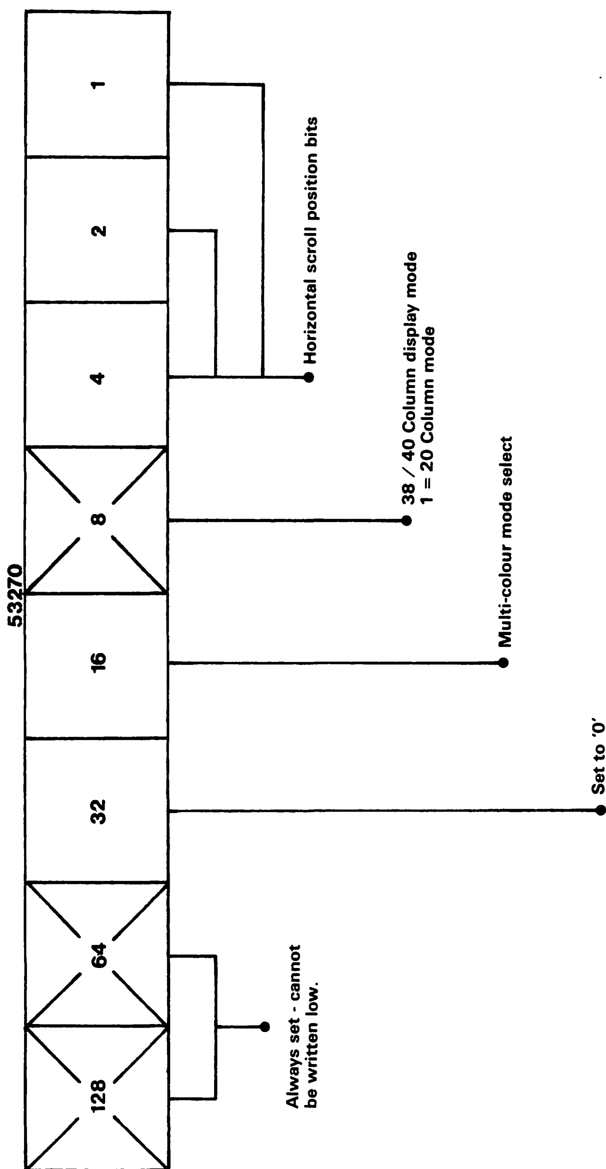


Figure 6.10 VIC 2 Register 22 Location 53270

Low Resolution Mode

In the default configuration, the screen character map is from 1024 - 2023 with the character sets appearing (to the VIC 2) from 4-6 and 6-8K. The foreground colour for each character cell is read from the 1000 nibble block of colour R.A.M. between 55296 and 56295.

If sprites are to be used, 64 byte blocks are used to define their shape. The 16K bank will accomodate up to 256 of these definition blocks.

As there are only 1000 screen positions, there are 24 'spare' locations in the 1K allocated for the screen memory and, on power up, 2040-7 are used to hold the data block referenced for each of the eight possible sprites.

Consequently, if the screen character map is moved, so too are the sprite pointers.

The position of the screen and character set may be moved freely within the 16K block. The screen map may start at the beginning of any one of the 16 1K blocks available - just place the number in the high order nibble of location 53272.

Note that whatever value is written to this register, when read the l.s. bit will appear set.

If use of the screen editor is required, the pointer it uses for the screen position, must also be updated. To do this place the page of the first location of the screen map in location 648 - i.e. if the screen was moved to 2048-3047 the pointer would be updated by placing 8 in location 648.

The character set used on power up, appears to the VIC 2 between 4 and 6K. This consists of 256 character definitions, each of eight bytes. If you want another 2K block to provide the character look up table, just select one of the eight possible positions by placing the number into the M.S. three bits of the low order nibble of location 53272.

The following program illustrates several of the features introduced so far:

```
                  ***************************
                  *                         *
                  * Example 1 : Text Screen *
                  *                         *
                  ***************************

                  10              ENT
                  20              ORG $1100

1100 78           30              SEI                Copy character set 1
1101 A9 33        40              LDA #51
1103 85 01        50              STA 1              Switch in character
1105 A2 09        60              LDX #9             ROM & set No. of
1107 A9 00        70              LDA #0             pages to copy
1109 85 FB        80              STA $FB
110B 85 FD        90              STA $FD            Set zero page pointer
110D A9 07        100             LDA #7
110F 85 FC        110             STA $FC
1111 A9 CF        120             LDA #$CF
1113 85 FE        130             STA $FE
1115 A0 FF        140             LDY #255
1117 C8           150 LOOP        INY
1118 D0 0A        160             BNE TRANSFER
111A E6 FE        170             INC $FE
111C E6 FC        180             INC $FC
111E CA           190             DEX
111F D0 03        200             BNE TRANSFER
1121 4C 2B 11     210             JMP END
1124 B1 FD        220 TRANSFER    LDA ($FD),Y        Copy character byte
1126 91 FB        230             STA ($FB),Y
1128 4C 17 11     240             JMP LOOP
112B A9 37        250 END         LDA #55
112D 85 01        260             STA 1              Switch out character
112F 58           270             CLI                        ROM

1130 A9 12        280             LDA #18
1132 8D 18 D0     290             STA 53272          Alter character table
1135 A9 00        300             LDA #0                   reference
1137 8D 20 D0     310             STA 53280
113A 8D 21 D0     320             STA 53281          Change screen colours

113D A2 05        330             LDX #5             Clear screen & set
113F A9 00        340             LDA #0             foreground colour
1141 85 FB        350             STA $FB
1143 85 FD        360             STA $FD            Set zero page pointers
1145 A9 03        370             LDA #3
1147 85 FC        380             STA $FC
1149 A9 D7        390             LDA #$D7
114B 85 FE        400             STA $FE
114D A0 FF        410             LDY #255
114F C8           420 LP          INY
1150 D0 0A        430             BNE TRF
1152 E6 FE        440             INC $FE
1154 E6 FC        450             INC $FC
```

```
1156 CA         460           DEX
1157 D0 03      470           BNE TRF
1159 4C 67 11   480           JMP END2
115C A9 20      490 TRF       LDA #32
115E 91 FB      500           STA ($FB),Y     Store space character
1160 A9 01      510           LDA #1
1162 91 FD      520           STA ($FD),Y     Store colour byte
1164 4C 4F 11   530           JMP LP

1167 A9 38      540 END2      LDA #56         Redefine @ character
1169 8D 00 08   550           STA 2048
116C 8D 01 08   560           STA 2049
116F A9 82      570           LDA #130
1171 8D 02 08   580           STA 2050
1174 A9 7C      590           LDA #124
1176 8D 03 08   600           STA 2051
1179 A9 10      610           LDA #16
117B 8D 04 08   620           STA 2052
117E A9 28      630           LDA #40
1180 8D 05 08   640           STA 2053
1183 8D 06 08   650           STA 2054
1186 8D 07 08   660           STA 2055

1189 A9 00      670           LDA #0          Place on screen
118B 8D F2 05   680           STA 1522

118E A9 E0      690           LDA #$E0        Set zero page pointer
1190 85 FB      700           STA $FB         to location of the
1192 A9 05      710           LDA #5          first column of the
1194 85 FC      720           STA $FC         row on which the
1196 A0 12      730           LDY #18         character will move

1198 A5 C5      740 LP1       LDA 197         Get key code
119A C9 40      750           CMP #64
119C F0 FA      760           BEQ LP1
119E C9 17      770           CMP #23
11A0 F0 2F      780           BEQ END3        If "X" finish
11A2 C9 3E      790           CMP #62
11A4 F0 14      800           BEQ LEFT        "Q" = Left
11A6 C9 09      810           CMP #9
11A8 D0 EE      820           BNE LP1

11AA C0 26      830 RIGHT     CPY #38         "W" = Right
11AC F0 EA      840           BEQ LP1         End of line check
11AE A9 20      850           LDA #32
11B0 91 FB      860           STA ($FB),Y     Delete character
11B2 C8         870           INY
11B3 A9 00      880           LDA #0
11B5 91 FB      890           STA ($FB),Y     Replace
11B7 4C C7 11   900           JMP WAIT

11BA C0 01      910 LEFT      CPY #1          End of line check
11BC F0 DA      920           BEQ LP1
11BE A9 20      930           LDA #32
11C0 91 FB      940           STA ($FB),Y     Delete character
```

```
11C2 88         950            DEY
11C3 A9 00      960            LDA #0
11C5 91 FB      970            STA ($FB),Y     Replace

11C7 A2 00      980 WAIT       LDX #0          Delay routine
11C9 E8         990 WAIT1      INX
11CA EA         1000           NOP
11CB EA         1010           NOP
11CC D0 FB      1020           BNE WAIT1
11CE 4C 98 11   1030           JMP LP1         Return to keyboard
11D1 60         1040 END3      RTS                  handler
```

How the program works

The first stages of an arcade style game on the text screen will require:

1. A character set to be downloaded and re-defined.
2. The screen cleared and the colour(s) set.
3. Key input analysis and movement routines.

The program in this section achieves these as follows:

1. Character set 1 is copied to between 2 and 4K.
2. The screen is cleared and all positions set to foreground white.
3. A character to represent the player is re-defined.
4. Position markers are set and the 'man' placed on the screen.
5. Key input and execution routines are provided.
6. Wait routine - very important!

This allows the player to move the 'man' from left to right using the 'Q' or 'W' keys - but beware: it's fast!

Although unlikely to elicit gasps of awe from Joe Public ('I could do that with one line of BASIC') it illustrates a couple of basic techniques and provides a foundation from which many games may be developed.

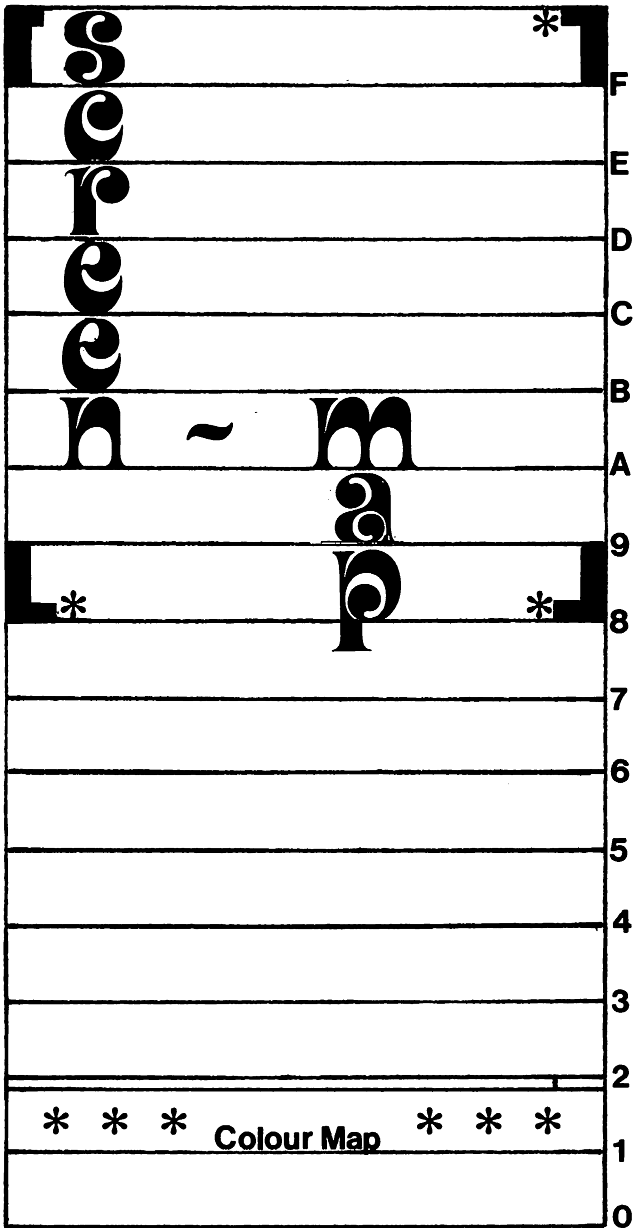


Figure 6.11 Example Bit map of configuration

In transferring the character set and setting up the screen, large blocks of data are either transferred or dumped. This is achieved by using a zero page pointer to the start of the block with the indirect-Y address format. After the Y register has been used to step through the locations in one page, the high order byte of the pointer is incremented ready for the next page.

The display of the 'man' is also accomplished using this address format.

A pointer to the leftmost character cell of the row used is placed in &FB and &FC, with the column position held in the Y register. End of line run off is prevented by a comparison of the Y register value with the most extreme position.

Keyboard handling is from examination of the physical key code number returned in location 197 ('Q':62 'W':9 'X':23).

On exiting from the routine a jumble of Qs and Ws will appear on screen - the keyboard queue.

Bit Map Mode

There are two display formats:

1. Standard 320*200 resolution
2. Multicolour 160*200 resolution

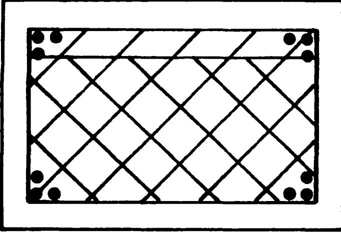
In the standard format, each of the 1000 screen cells no longer contains a character look up code but the foreground and background colour combination for that cell. The low order nibble holds the foreground and the high order nibble the background colour.

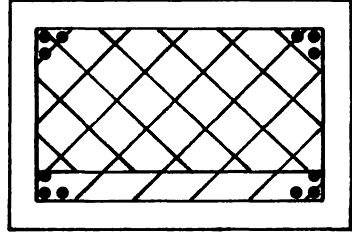
The screen itself is mapped using 8000 bytes with each controlling eight pixels - if a bit is set, it is displayed in the foreground colour otherwise it takes the background colour.

The bit mapped mode is entered by setting bit six of the VIC 2 register at location 53265. Note that the most significant bit of this register reads as the most significant bit of the raster count and therefore fluctuates.

The 8000 byte screen map may start either at the start of the 16K bank or on its 8K boundary. Selection is by the setting of the most significant bit of the low order nibble of location 53272.

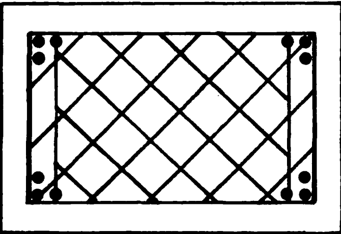
24 Row mode : Y Scroll bits = 0

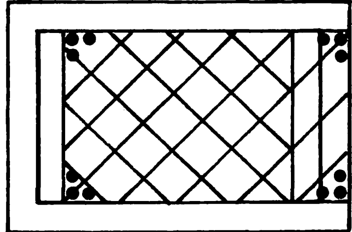




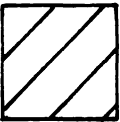
24 Row mode : Y Scroll bits = 7

38 Column mode : X Scroll bits = 0

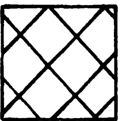




38 Column mode : X Scroll bits = 7



= Screen covered by border



= Visible Screen area

Figure 6.12 Screen scroll positions

The 1000 byte colour map is pointed to by the high order nibble of the same register - which is why on entering the bit map mode from default, it takes the same position as the text screen. Interestingly, the two character sets may be examined by entering the bit map mode without raising the screen map to 8K (POKE53265,59).

Multicolour Mode

This is entered by setting bit 6 of location 53270. Again 8000 bytes are used to map the screen but instead of each bit controlling one pixel, two bits are used to control a double size 'pixel'. The colour of each 'economy' size pixel is determined by the value of the two bits:

00 Screen background colour (53280)
01 H.O. nibble of the screen colour memory cell
10 L.O. nibble of the screen colour memory cell
11 The representative colour nibble (D800-DBFF).

Scrolling

This is simplified by two special abilities of the VIC 2 chip:

1. To move the whole screen one pixel position in any direction.
2. To expand the border to cloak the left and right hand columns and/or one row of the display.

We will first look at a left to right scroll. The first step is to shroud the left and right hand columns of the screen by setting bit 4 in register 53270.

To get the VIC 2 to move the entire screen one pixel left or right we use the three least significant bits in location 53270. These may take a value between 0 and 7 - default 0 (N.B. the two most significant bits of this register read as set, irrespective of any previous values written).

If we set bit 2, the screen is shunted one pixel right and the rightmost pixel column goes out of view with a 'new' column of pixels appearing on the left of the screen - from underneath the border.

We can repeat this shift seven times until the entire lefthand column that was previously covered has been brought into view. It is now time for us to do some work. A user routine shifts the screen memory so that the display moves eight pixels right. At the same time, a new left hand column is inserted and the scroll bits zeroed. The whole procedure is then repeated.

Vertical scrolling is slightly complicated by the VIC 2 only being able to

extend the border to cover one row of the screen. The pixel shunt is controlled by the three least significant bits of register 53265 (default value : 3). To scroll downwards, these are incremented from 0 to 7. Note that, as the scroll bit default value is three, when the 24 row mode is entered (by clearing bit 4 of location 532365) the border covers the top five and bottom three pixel rows. The complete upper row is covered by simultaneously zeroing the scroll bits.

Split Screen Effects

Here, part of the screen is displayed in bit mapped mode and the remainder in text mode. This is achieved by the use of raster interrupts. The non-border section of the display extends from raster 50-250. When the current raster position matches the value written to the raster register(s) the raster bit in the VIC 2 interrupt register (53273) is set. Thus, an interrupt may be 'pulled' at a certain raster position to change the mode in which the following lines are displayed. To produce the split screen effect with the top half in bit mapped mode we would write 150 to the raster register, location 53266 and enable raster interrupts by setting the l.s. bit of the interrupt mask register.

Consequently, when the raster count equalled 150 the interrupt would occur and the routine would swap the display mode to text and change the raster compare value to 40. The next raster interrupt would be used to swap back to bit mapped mode and re-write a compare value of 150. The final example at the end of this chapter features split screen graphics and if you are thinking of using the technique have a careful look - it is not straightforward.

Where possible, changes to the display are best made when the raster count is not between 50 and 250 as this avoids 'flicker'. Location 53266 is the read/write register for the least significant 8 bits of the raster count, with the ninth M.S. bit appearing as bit 8 of location 53265.

The following example 'MINESHAFT' illustrates many of the facilities mentioned.

```
                 ****************************
                 *                          *
                 * Example 2 : Bit map mode *
                 *                          *
                 *        'MINESHAFT'       *
                 *                          *
                 ****************************
```

```
                20               ENT
                30               ORG 24000

5DC0 48         40               PHA            INTERRUPT WEDGE
5DC1 8A         50               TXA
5DC2 48         60               PHA            Registers > Stack
5DC3 98         70               TYA
5DC4 48         80               PHA
5DC5 AD 1F D0   90               LDA 53279
5DC8 C9 01      100              CMP #1         Set flag on S-S
5DCA D0 02      110              BNE NXBIT         collision
5DCC 85 60      120              STA 96
5DCE AD 1E D0   130 NXBIT        LDA 53278
5DD1 F0 02      140              BEQ FINBIT     Set flag on S-B
5DD3 85 02      150              STA 2             collison
5DD5 A6 44      160 FINBIT       LDX 68
5DD7 E0 F8      170              CPX #248       Move 'Guard'
5DD9 D0 02      180              BNE MS2        sprite
5DDB A2 3C      190              LDX #60
5DDD E8         200 MS2          INX
5DDE 86 44      210              STX 68
5DE0 8E 02 D0   220              STX 53250
5DE3 68         230              PLA
5DE4 A8         240              TAY            Pull registers
5DE5 68         250              PLA
5DE6 AA         260              TAX
5DE7 68         270              PLA
5DE8 4C 31 EA   280              JMP $EA31      Goto interrupt
                                                    handler

                290              ORG 20000
4E20 20 2E 4E   300              JSR INITIAL     MAIN PROGRAM
4E23 20 90 4E   310              JSR SPRITES         CORE
4E26 A2 98      320              LDX #152
4E28 20 93 50   330              JSR INTWEDGE
4E2B 20 2A 4F   340              JSR KEY

4E2E A9 00      350 INITIAL      LDA #0
4E30 85 02      360              STA 2          Set S-S & S-B
4E32 85 60      370              STA 96         collision flags
4E34 A2 23      380              LDX #35        to zero
4E36 85 FD      390              STA $FD
```

```
4E38 A9 1E      400          LDA #30      Clear screen
4E3A 85 FE      410          STA $FE
4E3C A9 00      420          LDA #0
4E3E A0 FF      430          LDY #255
4E40 C8         440 LOOP     INY
4E41 D0 08      450          BNE TRF1
4E43 E6 FE      460          INC $FE
4E45 CA         470          DEX
4E46 D0 03      480          BNE TRF1
4E48 4C 50 4E   490          JMP END1
4E4B 91 FD      500 TRF1     STA ($FD),Y
4E4D 4C 40 4E   510          JMP LOOP
4E50 A9 30      520 END1     LDA #48
4E52 8D 11 D0   530          STA 53265
4E55 A9 18      540          LDA #24      Set screen position
4E57 8D 18 D0   550          STA 53272
4E5A A2 05      560 COLOUR   LDX #5
4E5C A9 00      570          LDA #0
4E5E 8D 16 D0   580          STA 53270    Set 38 Columns, bit
4E61 85 FB      590          STA $FB      map mode, 0 scroll
4E63 A9 03      600          LDA #3       bits
4E65 85 FC      610          STA $FC
4E67 A9 01      620          LDA #1
4E69 A0 FF      630          LDY #255
4E6B C8         640 LOOP2    INY
4E6C D0 08      650          BNE TRF2
4E6E E6 FC      660          INC $FC
4E70 CA         670          DEX
4E71 D0 03      680          BNE TRF2
4E73 4C 7B 4E   690          JMP END2
4E76 91 FB      700 TRF2     STA ($FB),Y
4E78 4C 6B 4E   710          JMP LOOP2
4E7B A0 40      720 END2     LDY #64      Put 0's in sprite
4E7D A9 03      730          LDA #3       data area
4E7F 85 FC      740          STA $FC
4E81 A9 40      750          LDA #$40
4E83 85 FB      760          STA $FB
4E85 A9 00      770          LDA #0
4E87 88         780 LOOP3    DEY
4E88 F0 05      790          BEQ END3
4E8A 91 FB      800          STA ($FB),Y
4E8C 4C 87 4E   810          JMP LOOP3
4E8F 60         820 END3     RTS

4E90 A9 38      830 SPRITES  LDA #56      S.Data 832-
4E92 8D 41 03   840          STA 833
4E95 8D 44 03   850          STA 836
4E98 8D 47 03   860          STA 839
4E9B 8D 4A 03   870          STA 842
4E9E A9 82      880          LDA #130
4EA0 8D 4D 03   890          STA 845
4EA3 8D 50 03   900          STA 848
4EA6 A9 7C      910          LDA #124
4EA8 8D 53 03   920          STA 851
```

```
4EAB 8D 56 03   930              STA 854
4EAE A9 10      940              LDA #16
4EB0 8D 59 03   950              STA 857
4EB3 8D 5C 03   960              STA 860
4EB6 A9 28      970              LDA #40
4EB8 8D 5F 03   980              STA 863
4EBB 8D 62 03   990              STA 866
4EBE 8D 65 03   1000             STA 869
4EC1 8D 68 03   1010             STA 872
4EC4 8D 6B 03   1020             STA 875
4EC7 8D 6E 03   1030             STA 878
4ECA A9 00      1040             LDA #0
4ECC 8D 40 03   1050             STA 832
4ECF A9 0D      1060             LDA #13
4ED1 8D F8 07   1070             STA 2040      Allocate block
4ED4 8D F9 07   1080             STA 2041      to sprites
4ED7 A9 03      1090             LDA #3
4ED9 8D 1D D0   1100             STA 53277     Expand horizontally
4EDC A9 08      1110             LDA #8
4EDE 8D 27 D0   1120             STA 53287     Set colours
4EE1 A9 02      1130             LDA #2
4EE3 8D 28 D0   1140             STA 53288
4EE6 A9 03      1150             LDA #3
4EE8 8D 15 D0   1160             STA 53269     Switch on
4EEB A0 00      1170             LDY #0
4EED A2 98      1180             LDX #152
4EEF 8E 00 D0   1190             STX 53248     Set X (player)
4EF2 8C 01 D0   1200             STY 53249     Set Y (player)
4EF5 A9 A0      1210             LDA #160
4EF7 8D 03 D0   1220             STA 53251     Set Y (guard)
4EFA A9 3C      1230             LDA #60
4EFC 8D 02 D0   1240             STA 53250     Set X (guard)
4EFF 85 44      1250             STA 68
4F01 A9 FF      1260             LDA #255      Place 'SCENERY'
4F03 A2 08      1270             LDX #8        on screen
4F05 CA         1280 DUMP        DEX
4F06 F0 21      1290             BEQ ENDUMP
4F08 9D 68 2B   1300             STA 11112,X
4F0B 9D E8 2C   1310             STA 11496,X
4F0E 9D 20 2D   1320             STA 11552,X
4F11 9D 4F 2D   1330             STA 11599,X
4F14 9D 50 2C   1340             STA 11344,X
4F17 9D 88 36   1350             STA 13960,X
4F1A 9D D0 36   1360             STA 14032,X
4F1D 9D 20 37   1370             STA 14112,X
4F20 9D 67 37   1380             STA 14183,X
4F23 9D B0 37   1390             STA 14256,X
4F26 4C 05 4F   1400             JMP DUMP
4F29 60         1410 ENDUMP      RTS

4F2A A5 02      1420 KEY         LDA 2         COLLISON CHECKS
4F2C F0 07      1430             BEQ COLL2
4F2E A9 00      1440             LDA #0
4F30 85 02      1450             STA 2
```

```
4F32 20 A9 4F   1460             JSR SCROLL
4F35 A5 60      1470 COLL2       LDA 96
4F37 F0 07      1480             BEQ KEY2
4F39 A9 00      1490             LDA #0
4F3B 85 60      1500             STA 96
4F3D 20 2A 50   1502             JSR SCROLL2

4F40 A5 C5      1510 KEY2        LDA 197       Analyse key input
4F42 C9 40      1520             CMP #64
4F44 D0 03      1530             BNE SIDE      If non goto DOWN
4F46 4C 71 50   1540             JMP DOWN
4F49 C9 17      1550 SIDE        CMP #23       If 'X' then end
4F4B D0 03      1560             BNE LABEL
4F4D 4C 66 50   1570             JMP THEND
4F50 C9 3E      1580 LABEL       CMP #62
4F52 F0 20      1590             BEQ LEFT
4F54 C9 09      1600 THER        CMP #9
4F56 D0 D2      1610             BNE KEY       If non valid key
4F58 AD 10 D0   1620 RIGHT       LDA 53264     return to start
4F5B F0 08      1630             BEQ NOTSET
4F5D E0 34      1640             CPX #52
4F5F F0 C9      1650             BEQ KEY       If at right of
4F61 E8         1660             INX           screen abort move
4F62 4C 8D 4F   1670             JMP DELAY
4F65 E0 FF      1680 NOTSET      CPX #255      Set MSB if on
4F67 D0 07      1690             BNE NOTYET    255 'boundery'
4F69 E8         1700             INX
4F6A EE 10 D0   1710             INC 53264
4F6D 4C 8D 4F   1720             JMP DELAY     Move right &
4F70 E8         1730 NOTYET      INX           goto delay routine
4F71 4C 8D 4F   1740             JMP DELAY
4F74 AD 10 D0   1750 LEFT        LDA 53264
4F77 F0 0F      1760             BEQ NSET
4F79 E0 00      1770             CPX #0
4F7B D0 07      1780             BNE MLEFT
4F7D CA         1790             DEX
4F7E CE 10 D0   1800             DEC 53264
4F81 4C 8D 4F   1810             JMP DELAY
4F84 CA         1820 MLEFT       DEX
4F85 4C 8D 4F   1830             JMP DELAY
4F88 E0 0C      1840 NSET        CPX #12
4F8A F0 9E      1850             BEQ KEY
4F8C CA         1860             DEX

4F8D 20 96 4F   1870 DELAY       JSR DLAY      Wait sequence
4F90 8E 00 D0   1880             STX 53248     Update X position
4F93 4C 2A 4F   1890             JMP KEY       Return to key scan

4F96 86 FF      1900 DLAY        STX $FF       Wait sequence
4F98 A2 00      1910             LDX #0
4F9A E8         1920 DELAY1      INX
4F9B EA         1930             NOP
4F9C EA                          NOP
4F9D EA                          NOP
```

```
4F9E EA         1940            NOP
4F9F EA                         NOP
4FA0 EA                         NOP
4FA1 EA         1950            NOP
4FA2 EA                         NOP
4FA3 EA                         NOP
4FA4 D0 F4      1960            BNE DELAY1
4FA6 A6 FF      1970 END        LDX $FF
4FA8 60         1980            RTS

4FA9 C8         1990 SCROLL     INY            Move player four
4FAA C8                         INY            pixel rows down
4FAB C8         2000            INY
4FAC C8                         INY
4FAD 8C 01 D0   2010            STY 53249
4FB0 98         2020            TYA
4FB1 48         2030            PHA
4FB2 8A         2040            TXA
4FB3 48         2050            PHA
4FB4 A9 0B      2060            LDA #11        Move screen 8
4FB6 8D 20 D0   2070            STA 53280      pixel rows up
4FB9 A9 3F      2080            LDA #$3F       then 8 down
4FBB A0 00      2090            LDY #0
4FBD 85 FB      2100            STA $FB
4FBF 85 FC      2110            STA $FC
4FC1 A9 3D      2120            LDA #$3D
4FC3 85 FE      2130            STA $FE
4FC5 A9 FF      2140            LDA #$FF
4FC7 85 FD      2150            STA $FD
4FC9 B1 FD      2160 SHIFT      LDA ($FD),Y
4FCB 91 FB      2170            STA ($FB),Y
4FCD A5 FB      2180            LDA $FB
4FCF C9 00      2190            CMP #0
4FD1 D0 09      2200            BNE CK1
4FD3 A9 FF      2210            LDA #$FF
4FD5 85 FB      2220            STA $FB
4FD7 C6 FC      2230            DEC $FC
4FD9 4C DE 4F   2240            JMP OTHER
4FDC C6 FB      2250 CK1        DEC $FB
4FDE A5 FD      2260 OTHER      LDA $FD
4FE0 C9 00      2270            CMP #0
4FE2 D0 0F      2280            BNE CK2
4FE4 A9 FF      2290            LDA #$FF
4FE6 85 FD      2300            STA $FD
4FE8 C6 FE      2310            DEC $FE
4FEA A5 FE      2320            LDA $FE
4FEC C9 1F      2330            CMP #$1F
4FEE D0 D9      2340            BNE SHIFT
4FF0 4C F8 4F   2350            JMP RSB
4FF3 C6 FD      2360 CK2        DEC $FD
4FF5 4C C9 4F   2370            JMP SHIFT
4FF8 A9 00      2380 RSB        LDA #0
4FFA A0 00      2390            LDY #0
4FFC 85 FB      2400            STA $FB
```

```
4FFE A9 20       2410          LDA #$20
5000 85 FC       2420          STA $FC
5002 A9 40       2430          LDA #$40
5004 85 FD       2440          STA $FD
5006 A9 21       2450          LDA #$21
5008 85 FE       2460          STA $FE
500A B1 FD       2470 SHFT     LDA ($FD),Y
500C 91 FB       2480          STA ($FB),Y
500E E6 FB       2490          INC $FB
5010 D0 02       2500          BNE CH1
5012 E6 FC       2510          INC $FC
5014 E6 FD       2520 CH1      INC $FD
5016 D0 02       2530          BNE CH2
5018 E6 FE       2540          INC $FE
501A A5 FC       2550 CH2      LDA $FC
501C C9 3F       2560          CMP #$3F
501E D0 EA       2570          BNE SHFT
5020 68          2580 NREND    PLA
5021 AA          2590          TAX
5022 68          2600          PLA
5023 A8          2610          TAY
5024 A9 00       2620          LDA #0
5026 8D 20 D0    2630          STA 53280      Reset border
5029 60          2640          RTS


502A C8          2650 SCROLL2  INY            Move player four
502B C8                        INY            pixel rows down
502C C8          2660          INY
502D C8                        INY
502E 8C 01 D0    2670          STY 53249
5031 98          2680          TYA
5032 48          2690          PHA
5033 8A          2700          TXA
5034 48          2710          PHA
5035 A0 00       2720          LDY #0         Scroll screen 7
5037 A2 08       2730          LDX #8         pixels either way
5039 C8          2740 UPP      INY
503A CA          2750          DEX
503B F0 12       2760          BEQ RO
503D 8C 16 D0    2770          STY 53270
5040 20 96 4F    2780          JSR DLAY
5043 20 96 4F    2790          JSR DLAY
5046 20 96 4F    2800          JSR DLAY
5049 20 96 4F    2810          JSR DLAY
504C 4C 39 50    2820          JMP UPP
504F A2 08       2830 RO       LDX #8
5051 88          2840 DOW      DEY
5052 CA          2850          DEX
5053 F0 0C       2860          BEQ FINI
5055 8C 16 D0    2870          STY 53270
5058 20 96 4F    2880          JSR DLAY
505B 20 96 4F    2890          JSR DLAY
505E 4C 51 50    2900          JMP DOW
5061 68          2910 FINI     PLA
```

```
5062 AA          2920            TAX
5063 68          2930            PLA
5064 A8          2940            TAY
5065 60          2950            RTS

5066 A9 1B       2960 THEND      LDA #27         Exit sequence
5068 8D 11 D0    2970            STA 53265
506B A9 15       2980            LDA #21
506D 8D 18 D0    2990            STA 53272
5070 60          3000            RTS

5071 EE 01 D0    3010 DOWN       INC 53249       Move player down
5074 C8          3020            INY             one pixel row. If
5075 AD 01 D0    3030            LDA 53249       at bottom of screen
5078 C9 EA       3040            CMP #234        reset to the top
507A D0 11       3050            BNE FINISH
507C A9 3C       3060            LDA #60
507E 8D 01 D0    3070            STA 53249
5081 A8          3080            TAY
5082 A5 A2       3090            LDA 162
5084 8D 00 D0    3100            STA 53248
5087 AA          3110            TAX
5088 A9 00       3120            LDA #0
508A 8D 10 D0    3130            STA 53264
508D 20 96 4F    3140 FINISH     JSR DLAY
5090 4C 2A 4F    3150            JMP KEY

5093 78          3160 INTWEDGE   SEI
5094 A9 C0       3170            LDA #$C0        Replace IRQ vector
5096 8D 14 03    3180            STA 788         to point to wedge
5099 A9 5D       3190            LDA #$5D        (lines 40-280) at
509B 8D 15 03    3200            STA 789         24000
509E 58          3210            CLI
509F 60          3220            RTS
```

The program uses a bit mapped screen and sprite graphics with collision (player to 'guard') initiates a left to right and right to left scroll of the screen, whilst a sprite to background collision causes the screen to be shifted one row up and one row down.

The player's sprite is continuously moved down the screen with keyboard control by the 'Q' and 'W' keys for left and right movement. The guard sprite is moved from left to right by the interrupt wedge.

The wedge is located from address 24000. After completing its various functions, control is directed back to the interrupt handler at &EA31. The interrupt route is diverted to the wedge by altering the vector at locations 788 and 789 (lines 3160-3220).

The main program is assembled from location 20000 and consists of a core which calls the principle subroutines -

1. INITIAL
2. SPRITES
3. INTWEDGE
4. KEY

This format is fairly common as it allows new sections to be added or old routines removed with ease.

The labels used are, where possible, indicative of function. However, it is probably advisable to use short sequenced labels in your own routines as, with long programs, label replication can be a problem.

If you are not familiar with an assembler, lines 20 and 30 may look a little strange. The ENT instruction is used to tell the assembler to start from the next line whilst the ORG statement tells it where to place the final code.

The last example, 'PRONG' also uses an interrupt wedge to flag sprite-sprite collisions. However, the bottom five lines are displayed in text to allow the score etc. to be maintained. To achieve this, the raster interrupts are enabled to switch between the screen modes and this is the main function of the interrupt wedge. The VIC interrupt register is checked to see if the interrupt was pulled by a raster match and if so, the change is made before exiting. As the initial section of the 64's interrupt routine saves the general registers to the stack, in order to get the correct return location, these must be pulled before the RTI is used. The program also demonstrates a simple score/rest capability.

```
                  ***********
                 *************
                ****       ****
               ***** PRONG *****
                ****       ****
                 *************
                  ***********
```

```
               10             ENT
               20             ORG 24000

5DC0 48        30             PHA          Interrupt wedge
5DC1 8A        40             TXA
5DC2 48        50             PHA          Registers > stack
5DC3 98        60             TYA
5DC4 48        70             PHA
5DC5 AD 19 D0  80             LDA 53273
5DC8 29 01     90             AND #1       Raster interrupt ?
5DCA F0 36     100            BEQ COLLCK
5DCC 8D 19 D0  110            STA 53273    Clear flag
5DCF AD 11 D0  120            LDA 53265
5DD2 29 20     130            AND #32
5DD4 F0 12     140            BEQ TOBITMAP Check display mode
5DD6 A9 28     150            LDA #40
5DD8 8D 12 D0  160            STA 53266    Set new raster compare
5DDB A9 14     170            LDA #20              value
5DDD 8D 18 D0  180            STA 53272
5DE0 A9 1B     190            LDA #27
5DE2 8D 11 D0  200            STA 53265    Enter text mode
5DE5 4C F7 5D  210            JMP PULL
5DE8 A9 D0     220 TOBITMAP   LDA #208
5DEA 8D 12 D0  230            STA 53266    Set new raster compare
5DED A9 1C     240            LDA #28              value
5DEF 8D 18 D0  250            STA 53272
5DF2 A9 3B     260            LDA #59
5DF4 8D 11 D0  270            STA 53265    Enter bit mapped mode
5DF7 68        280 PULL       PLA
5DF8 A8        290            TAY
5DF9 68        300            PLA
5DFA AA        310            TAX          Pull registers *2
5DFB 68        320            PLA
5DFC 68        330            PLA
5DFD A8        340            TAY
5DFE 68        350            PLA
5DFF AA        360            TAX
5E00 68        370            PLA
5E01 40        380            RTI
5E02 AD 1E D0  390 COLLCK     LDA 53278    Set flag on S-S collision
5E05 F0 02     400            BEQ NXBIT
5E07 85 02     410            STA 2
5E09 EE 02 D0  420 NXBIT      INC 53250    Move the 'Prong'
5E0C 68        430 LASTBIT    PLA
```

```
5E0D A8         440             TAY
5E0E 68         450             PLA
5E0F AA         460             TAX
5E10 68         470             PLA
5E11 4C 31 EA   480             JMP $EA31     Goto interrupt handler


                490 'CORE'
                500             ORG 20000

4E20 20 2C 4E   510             JSR INITIAL   Main program core
4E23 20 B5 4E   520             JSR SPRITES
4E26 20 59 4F   530             JSR START
4E29 20 83 4F   540             JSR KEY

4E2C A9 00      550 INITIAL     LDA #0        Clear collision flag
4E2E 85 02      560             STA 2
4E30 A2 20      570             LDX #32
4E32 85 FD      580             STA $FD
4E34 A9 1E      590             LDA #30
4E36 85 FE      600             STA $FE
4E38 A9 00      610             LDA #0
4E3A A0 FF      620             LDY #255
4E3C C8         630 LOOP        INY
4E3D D0 08      640             BNE TRF
4E3F E6 FE      650             INC $FE
4E41 CA         660             DEX
4E42 D0 03      670             BNE TRF
4E44 4C 4C 4E   680             JMP END1
4E47 91 FD      690 TRF         STA ($FD),Y     Clear bit mapped
4E49 4C 3C 4E   700             JMP LOOP             screen
4E4C A9 3C      710 END1        LDA #$3C
4E4E 85 FB      720             STA $FB
4E50 A9 03      730             LDA #3
4E52 85 FC      740             STA $FC
4E54 A9 00      750             LDA #0
4E56 A0 00      760             LDY #0          Clear sprite data
4E58 E6 FB      770 LP1         INC $FB              area
4E5A F0 05      780             BEQ END2
4E5C 91 FB      790             STA ($FB),Y
4E5E 4C 58 4E   800             JMP LP1
4E61 A9 FF      810 END2        LDA #$FF
4E63 85 FB      820             STA $FB
4E65 E6 FB      830 LP2         INC $FB
4E67 A5 FB      840             LDA $FB
4E69 C9 20      850             CMP #$20
4E6B D0 06      860             BNE TRF1
4E6D A5 FC      870             LDA $FC
4E6F C9 07      880             CMP #7
4E71 F0 0D      890             BEQ END3
4E73 A5 FB      900 TRF1        LDA $FB
4E75 D0 02      910             BNE TRF2
4E77 E6 FC      920             INC $FC
4E79 A9 01      930 TRF2        LDA #1
```

```
4E7B 91 FB      940          STA ($FB),Y    Set bit mapped screen
4E7D 4C 65 4E   950          JMP LP2              colours
4E80 A0 00      960 END3     LDY #0
4E82 A9 20      970 TRF3     LDA #32
4E84 91 FB      980          STA ($FB),Y    Clear a section of the
4E86 C8         990          INY                 text screen
4E87 C0 C8      1000         CPY #200
4E89 D0 F7      1010         BNE TRF3
4E8B A0 00      1020         LDY #0
4E8D A9 1F      1030         LDA #31
4E8F 85 FB      1040         STA $FB
4E91 A9 DB      1050         LDA #$DB
4E93 85 FC      1060         STA $FC
4E95 C8         1070 TRF4    INY
4E96 C0 C8      1080         CPY #200
4E98 F0 07      1090         BEQ END4
4E9A A9 01      1100         LDA #1
4E9C 91 FB      1110         STA ($FB),Y   Set text foreground
4E9E 4C 95 4E   1120         JMP TRF4             colour
4EA1 A9 30      1130 END4    LDA #48
4EA3 8D 84 07   1140         STA 1924      Place score on screen
4EA6 8D 85 07   1150         STA 1925
4EA9 8D 86 07   1160         STA 1926
4EAC A9 00      1170         LDA #0
4EAE 8D 20 D0   1180         STA 53280     Set global colours
4EB1 8D 21 D0   1190         STA 53281
4EB4 60         1200         RTS

4EB5 A9 38      1210 SPRITES LDA #56       Define the players
4EB7 8D 41 03   1220         STA 833            sprite
4EBA 8D 44 03   1230         STA 836
4EBD 8D 47 03   1240         STA 839
4EC0 8D 4A 03   1250         STA 842
4EC3 A9 82      1260         LDA #130
4EC5 8D 4D 03   1270         STA 845
4EC8 8D 50 03   1280         STA 848
4ECB A9 7C      1290         LDA #124
4ECD 8D 53 03   1300         STA 851
4ED0 8D 56 03   1310         STA 854
4ED3 A9 10      1320         LDA #16
4ED5 8D 59 03   1330         STA 857
4ED8 8D 5C 03   1340         STA 860
4EDB A9 28      1350         LDA #40
4EDD 8D 5F 03   1360         STA 863
4EE0 8D 62 03   1370         STA 866
4EE3 8D 65 03   1380         STA 869
4EE6 8D 68 03   1390         STA 872
4EE9 8D 6B 03   1400         STA 875
4EEC 8D 6E 03   1410         STA 878
4EEF A9 0D      1420         LDA #13
4EF1 8D F8 07   1430         STA 2040       Allocate to sprite
4EF4 A9 08      1440         LDA #8
4EF6 8D 27 D0   1450         STA 53287      Set the colour
4EF9 A0 00      1460         LDY #0
4EFB A2 14      1470         LDX #20
```

```
4EFD A9 84       1480          LDA #$84
4EFF 85 FB       1490          STA $FB
4F01 A9 03       1500          LDA #3
4F03 85 FC       1510          STA $FC
4F05 A9 FF       1520          LDA #255
4F07 91 FB       1530 LP3      STA ($FB),Y     Define the 'Prong'
4F09 C8          1540          INY                   sprite
4F0A C8          1550          INY
4F0B C8          1560          INY
4F0C CA          1570          DEX
4F0D D0 F8       1580          BNE LP3
4F0F A9 FE       1590          LDA #254
4F11 8D 88 03    1600          STA 904
4F14 8D 8B 03    1610          STA 907
4F17 8D 9D 03    1620          STA 925
4F1A 8D A0 03    1630          STA 928
4F1D 8D B5 03    1640          STA 949
4F20 8D B8 03    1650          STA 952
4F23 A9 04       1660          LDA #4
4F25 8D 85 03    1670          STA 901
4F28 8D 8E 03    1680          STA 910
4F2B 8D 9A 03    1690          STA 922
4F2E 8D A3 03    1700          STA 931
4F31 8D B2 03    1710          STA 946
4F34 8D BB 03    1720          STA 955
4F37 A9 0E       1730          LDA #14
4F39 8D F9 07    1740          STA 2041         Allocate to sprite
4F3C A9 06       1750          LDA #6
4F3E 8D 28 D0    1760          STA 53288        Set the colour
4F41 A9 03       1770          LDA #3
4F43 8D 1D D0    1780          STA 53277        Expand horizontally
4F46 8D 15 D0    1790          STA 53269        Activate
4F49 A9 8C       1800          LDA #140
4F4B 8D 00 D0    1810          STA 53248        Position on screen
4F4E A9 78       1820          LDA #120
4F50 8D 01 D0    1830          STA 53249
4F53 A5 A2       1840          LDA 162
4F55 8D 03 D0    1850          STA 53251
4F58 60          1860          RTS

4F59 A9 3B       1870 START    LDA #59          Enter the bit mapped
4F5B 8D 11 D0    1880          STA 53265                mode
4F5E A9 1C       1890          LDA #28
4F60 8D 18 D0    1900          STA 53272
4F63 A9 D0       1910          LDA #208
4F65 8D 12 D0    1920          STA 53266        Set the raster
4F68 20 76 4F    1930          JSR INTWEDGE     compare value & alter
4F6B 78          1932          SEI              the IRQ vector
4F6C AD 1A D0    1940          LDA 53274
4F6F 09 01       1950          ORA #1           Enable raster
4F71 8D 1A D0    1960          STA 53274        interrupts
4F74 58          1970          CLI
4F75 60          1980          RTS

4F76 78          1990INTWEDGE SEI
```

```
4F77 A9 C0      2000          LDA #$C0
4F79 8D 14 03   2010          STA 788
4F7C A9 5D      2020          LDA #$5D
4F7E 8D 15 03   2030          STA 789
4F81 58         2040          CLI
4F82 60         2050          RTS

4F83 AE 02 D0   2060 KEY      LDX 53250     Move the 'prong'
4F86 E8         2070          INX
4F87 D0 0B      2080          BNE KE
4F89 20 CB 4F   2090          JSR UPDATE    Update the score
4F8C 20 CB 4F   2100          JSR UPDATE
4F8F A5 A2      2110          LDA 162
4F91 8D 03 D0   2120          STA 53251     Set a random
4F94 8E 02 D0   2130 KE       STX 53250       Y value
4F97 20 17 50   2140          JSR DLAY
4F9A A5 02      2150          LDA 2
4F9C F0 03      2160          BEQ KEY1
4F9E 20 22 50   2170          JSR COLLISION If a collision is
4FA1 A5 C5      2180 KEY1     LDA 197        flagged, clear & reset
4FA3 C9 40      2190          CMP #64        the score
4FA5 F0 DC      2200          BEQ KEY
4FA7 C9 3E      2210          CMP #64       Get & execute key input
4FA9 F0 12      2220          BEQ UP
4FAB C9 09      2230          CMP #9
4FAD D0 D4      2240          BNE KEY
4FAF AC 01 D0   2250 DOWN     LDY 53249
4FB2 C0 C0      2260          CPY #192
4FB4 F0 CD      2270          BEQ KEY
4FB6 C8         2280          INY
4FB7 8C 01 D0   2290          STY 53249
4FBA 4C 83 4F   2300          JMP KEY
4FBD AC 01 D0   2310 UP       LDY 53249
4FC0 C0 32      2320          CPY #50
4FC2 F0 BF      2330          BEQ KEY
4FC4 88         2340          DEY
4FC5 8C 01 D0   2350          STY 53249
4FC8 4C 83 4F   2360          JMP KEY

4FCB AE 3F 03   2370 UPDATE   LDX 831       Update the score
4FCE E8         2380          INX           (locations 829-831)
4FCF E0 0A      2390          CPX #10
4FD1 D0 27      2400          BNE SHOW
4FD3 A9 00      2410          LDA #0        Display update on
4FD5 8D 3F 03   2420          STA 831       the text screen
4FD8 AE 3E 03   2430          LDX 830
4FDB E8         2440          INX
4FDC E0 0A      2450          CPX #10
4FDE D0 11      2460          BNE SHOW1
4FE0 8D 3E 03   2470          STA 830
4FE3 AE 3D 03   2480          LDX 829
4FE6 E8         2490          INX
4FE7 E0 0A      2500          CPX #10
4FE9 D0 0C      2510          BNE SHOW2
4FEB 8D 3D 03   2520          STA 829
```

```
4FEE 4C FD 4F  2530              JMP SHOW4
4FF1 8E 3E 03  2540 SHOW1        STX 830
4FF4 4C FD 4F  2550              JMP SHOW4
4FF7 8E 3D 03  2560 SHOW2        STX 829
4FFA 8E 3F 03  2570 SHOW         STX 831
4FFD 18        2580 SHOW4        CLC
4FFE A9 30     2590              LDA #48
5000 6D 3D 03  2600              ADC 829
5003 8D 84 07  2610              STA 1924
5006 A9 30     2620              LDA #48
5008 6D 3E 03  2630              ADC 830
500B 8D 85 07  2640              STA 1925
500E A9 30     2650              LDA #48
5010 6D 3F 03  2660              ADC 831
5013 8D 86 07  2670              STA 1926
5016 60        2680              RTS

5017 A2 00     2690 DLAY         LDX #0       Delay routine
5019 E8        2700 DLAY1        INX
501A EA        2710              NOP
501B D0 FC     2720              BNE DLAY1
501D E8        2730 DLAY2        INX
501E EA        2740              NOP
501F D0 FC     2750              BNE DLAY2
5021 60        2760              RTS

5022 A9 00     2770 COLLISION LDA #0          Clear collision flag
5024 85 02     2780              STA 2
5026 A9 EF     2790              LDA #239
5028 2D 11 D0  2800              AND 53265
502B 8D 11 D0  2810              STA 53265    Blank the screen
502E 20 17 50  2820              JSR DLAY
5031 A9 01     2830              LDA #1
5033 8D 20 D0  2840              STA 53280
5036 20 17 50  2850              JSR DLAY
5039 A9 10     2860              LDA #16
503B 0D 11 D0  2870              ORA 53265
503E 8D 11 D0  2880              STA 53265
5041 20 17 50  2890              JSR DLAY
5044 A9 00     2900              LDA #0
5046 8D 20 D0  2910              STA 53280    Change the border
5049 8D 3D 03  2920              STA 829
504C 8D 3E 03  2930              STA 830
504F 8D 3F 03  2940              STA 831
5052 A9 30     2950              LDA #48
5054 8D 84 07  2960              STA 1924     Reset the score
5057 8D 85 07  2970              STA 1925
505A 8D 86 07  2980              STA 1926
505D 60        2990              RTS
```

CHAPTER 7

The Commodore 64 R.O.M.

Overview. Program Storage. Input / Output. The Kernal.

Overview

On power up the top 8K of memory is occupied by the Kernal operating system. This is primarily concerned with interrupt processing i.e. input / output operations. Further down from locations 40960 to 49151 is the 8K BASIC interpreter. To pass control to the interpreter from a machine code routine, use the indirect form of the JMP instruction to location &A000.

Program Storage

As a BASIC program is entered, it is stored not as it is written, but in a contracted form with each keyword replaced by a code byte or token. In addition, each line is prefaced by two pairs of bytes. The second pair holds the line number, the first the address of the following line. Lines are separated by a byte set to zero, with two further bytes marking the program end.

A simple anti-list technique may be employed which exploits this structure. A note is made of the value taken by the two bytes following the initial end of line marker. A dummy end to the program is then created if zero is poked into each.

The list command is now limited to the first line.

To run the program, two poke statements are incorporated in the first line. Their purpose is to return the original value to the dummy end bytes. In turn, these statements may be made opaque by poking a series of deletes.

As a line of a BASIC program is entered, the token substituted for each keyword is found by a comparison to the list held between locations 41118 to 41864. If the user alters the comparison codes in the list, then until reset,the command must be entered in that form. This operation first requires the interpreter ROM to be switched out after it has been copied into the underlying RAM.

Variables

At the end of chapter 4, we touched on the storage requirments of each type of variable - integers record the actual value in 2 bytes and the floating point format in 5.

In BASIC, the variables declared or calculated are stored just above the program. As this has no fixed length, the storage area is moved as necessary. The start of the variable block is pointed to by two bytes in locations 45 and 46. A pointer to the next free byte is held in locations 47 and 48.

Non-array variables are stored in seven byte elements - whether integer, floating point or string. Whilst, for numeric data, this contains the data representation, the string element only contains a pointer to the string's location.

Byte	*1*	*2*	*3*	*4*	*5*	*6*	*7*
Integer	NAME +128	NAME +128	H.O.B.	L.O.B.	0	0	0
F.P.	NAME	NAME	Exponent + 128	Mantissa (1st bit is the sign)			
String	NAME	NAME +128	Length	Address L.O.B.H.O.B.		0	0

Figure 7.2 Simple variable formats

Each block of seven bytes starts with the ASC II codes of the first two characters of the variable name. The 64 differentiates between the variable types by adding 128 to each code for an integer, or to the second to indicate a string.

Of the remaining 5 bytes, in the integer form the three low order bytes are unused. Floating point values are represented in the form given in chapter 4 albeit the M.S. bit of the mantissa is used to flag the sign.

If the floating point accumulators are utilized, a separate byte is used to flag the variable sign. A negative argument is indicated by &FF and a zero or positive argument by &00.

String storage is at the address given by the 4th & 5th bytes in the element. The string length is held in the third byte. Bytes 6 and 7 are not used.

One advantage of this method of storage is that the variables may be cleared by the simple expedient of resetting the pointers.

It is important to note that if Memtop is altered, the string storage pointer (locations 51 and 52) must also be amended.

Clearly, no memory saving is derived from the use of integer as opposed to F.P. variables.

Input/Output

On the 64 this is most easily achieved by calling the appropriate kernal routines. These follow the same logic as the BASIC file handling commands.

To open a file, three routines are used: SETLFS, SETNAM and OPEN.

Initially, a file is set up by placing the logical file number in the X register, the device number in the A. and command (secondary address) in the Y register. The SETLFS routine, vector &FFBA, is then called. If no command is necessary, the Y register must be loaded with 255.

To use a file name, the string is first placed in memory & allocated using the SETNAM routine, vector &FFBD. Prior to the call, the address of the first character in the name is placed in the X & Y registers (low byte in the X register) & its length in the accumulator. It is still necessary to call this routine when no file name is used, with the accumulator set to zero.

Finally, the OPEN routine, vector &FFC0, is called.

It is now a straight-forward matter to transfer a complete file to or from the 64. To input (RS-232 excepted) to a specific position in memory, the address is placed in the X & Y registers, the accumulator is loaded with 0 and the LOAD routine, vector &FFD5, called. On completion, this procedure places the top transfer location in the X & Y registers. To load data at the location from which it was saved, a command of 1 in SETLFS, is necessary.

Saving a file is slightly less straightforward. The accumulator is loaded with the zero page location of a two byte pointer to the start of the file. The routine SAVE, vector &FFD8, is then called with the file end address in the X & Y registers (low order in the X).

Again the RS-232 interface may not be used with this routine.

If not saving to tape, a file name is obligatory.

Joysticks

The attachment and use of one or two digital joysticks is easily accomplished.

Each nine pin port uses one of the pair of registers that return the column / row position on a key press. Port A appears at location 56320 and Port B at 56321.

The joystick will only affect the five low order bits of each register. Figure 7.1 illustrates the configuration.

Action	*Fire*	*Right*	*Left*	*Down*	*Up*
Bit	5	4	3	2	1

Figure 7.1 C.I.A. 1 Data Port - Joystick options

Closing one of the direction or fire switches holds the respective bit low.

When few options are active, the analysis is best achieved by ANDing the register and a mask byte. However, in general use the most effective method is to inspect the carry bit. This is loaded with each bit in turn, using the LSR operation.

The User Port

This is configured as two rows of 12 pins / lines. Pins 15-22 are mapped to the C.I.A. 2 data port B. Service is requested using pin 14, which is mapped to the FLAG bit in its interrupt data register. The C.I.A. interrupt registers were discussed in chapter 6. Briefly, a NMI from this line is enabled by setting the equivalent bit in the mask register. The line is also used in conjunction with line 23 for handshaking procedures.

Pin 3 may be used to initiate a RESET if grounded - the sequence executed on power up. Interestingly, although restoring the system to default values, RESET will not clear a program from memory. Pins 1 and 24 are ground.

Serial Bus

Externally, this appears as a six pin DIN socket. Appendix J lists the format. Note that pin 6 is unused and pin 2 is ground. Actual data transfer is on line 5. A device may request attention via line one, mapped as the FLAG 1 bit in the C.I.A.1 interrupt data register.

When the 64 is to send data, it first checks the target device is on the bus. Confirmation is sought by bringing line 3 low. Line 4 is used for transmission timing.

In the previous section, we considered the I/O of complete files. To facilitate the transmission of single bytes on the serial bus, the kernal contains a set of specific routines.

Plainly, as a single line is used for transmission, only one device may transmit but upto five are able to listen.

To make a device talk i.e. put data on the bus, the device number is placed in the accumulator and the routine TALK, vector &FFB4 called. One byte is then input using the ACPTR routine, vector &FFA5. The data is deposited in the accumulator.

To transmit to a device, use the LISTEN procedure, vector &FFB1, with the device number in the accumulator. As with the TALK routine, the device number must lie in the range 0-31.

The CIOUT routine, vector &FFA8, manages byte transfer - the accumulator is used as the data source. To send a secondary address in the TALK or LISTEN modes, the routines TKSA and SECOND, respectively, are used prior to data transfer.

At the end of transmission, all devices on the bus may be told to stop listening with the UNLSN routine. The UNTALK routine halts the enabled TALK device.

Finishing off

Three routines are involved in the closure of a file or files: CLALL, CLOSE and CLRCHN. To close all open files and restore input and output to the defaultg channels - keyboard and screen - use CLALL, vector &FFE7. To close a particular file, place the logical file number in the accumulator and call CLOSE, vector &FFC3. CLRCHIN is used by CLALL to set to default the I/O channels.

Other Kernal Routines

Of the fourteen kernal routines not yet discussed, three relate to screen use, three to the system clock and the rest to system vectors/initialization procedures.

The System Clock

Locations 160-162 contain the system clock. This is incremented by the UDTIM routine called each 1/60th of a second. On power up the clock is set to zero. It is worth noting that the clock is not updated during input/output operations.

It is read using the RDTIM routine, vector &FFDE, which places the M.S. byte in the accumulator, the L.S. byte in the Y register and location 161 in the X register. The time is set by placing the required values in the general registers before calling the SETTIM routine, vector &FFDB.

The System Clock should not be confused with the real time clocks integral to each C.I.A. chip, which appear in memory from locations 56484-56587 and 56327-56331.

1/10 seconds	*Seconds*	*Minutes*	*AM/PM / Hours*
56484	56485	56486	56487
56328	56329	56330	56331

Figure 7.3 C.I.A. Real Time Clock Registers

The time is represented in binary coded decimal. A read of either hour register will halt the clock which is started again by a read of the 1/10 second register. Either clock may be used to initiate an interrupt by the use of the alarm facility. Each set of registers may also hold an alarm setting. A write to the registers will set the time if the M.S. bit of the C.I.A. control register is low, otherwise it is taken as an alarm setting.

Screen Routines

Probably the most useful of these is PLOT, vector &FFF0. Not unusually, the carry flag determines which of two functions is performed. If set, the current cursor position is placed in the index registers, otherwise the cursor takes the position they contain. The X register represents the Y co-ordinate and the Y register the X co-ordinate!

To enable a program to perform self-adjustment to changes in text screen format, incorporate the SCREEN routine, vector &FFED. This places the current number of row and column positions in the X & Y registers.

The CINT procedure initializes the V.I.C. 2 chip & screen editor for normal use. Its primary use is as part of the initialization sequence forced by the R.O.M. cartridges. The vector is &FF81.

System procedures

The upper and lower limits of the area available to BASIC are reset using MEMTOP & MEMBOT. The vectors are at &FF99 and &FF9C respectively. If the carry flag is set, either routine will return the pointer, not reset it. In either case the address is communicated using the X & Y registers (low order byte in the X register).

The IOBASE routine is, again, concerned with the forward compatibility of programs. It places the base address from which the I/O devices appear in memory, in the index registers Consequently, if the user's routine handles all I/O by calls relative to this address then no compatibility problems should be encountered with future versions of the 64.

On power up, the initialization of R.A.M., cassette buffer and screen location are performed by the RAMTAS routine, vector &FF87.

The SETMSG routine will switch on or off the system and error messages. Both sets may be negated by loading the accumulator with 0 prior to the the call, vector &FF90. If the accumulator has the M.S.B. set, the error messages are enabled, with the adjacent bit controlling system messages.

The three remaining routines are concerned with I/O vectors and operational status. RESTOR, vector &FF8A, resets all the system vectors. VECTOR, &FF8D, will if the carry is set, list the vectors at the address held in the X and Y registers. If the carry is clear, it will insert the vectors in the table pointed to by the address held in the index registers.

Finally, the READST routine is used at the end of a I/O operation. No preparatory values are necessary prior to the call, vector &FFB7. The value returned in the accumulator will indicate if a particular error or condition arose. These are listed in appendix K.

CHAPTER 8

ODDS AND ENDS

Cassette handling. Program protection. More resident routines.

Cassette Handling

SAVE XY$ / SAVE "XYZ"
Save the program between the pointers at 43-4 and 45-6

SAVE "XYZ",1,1
As above - albeit the program will load in at the location from which it was saved.

SAVE "XYZ",1,2
As for SAVE "XYZ" but an end of tape marker is added.

LOAD XY$ / LOAD "XYZ"
The file XY$ is loaded in from location 2048 onwards. If it was saved with a secondard address of 1, the program is loaded to the location from which it was saved.

LOAD "XYZ",1,1
Loads file "XYZ" in at the location from which it was originally saved.

10 LOAD "XYZ",1,1
File XYZ is loaded into memory from the location from which it was saved. The program at the location given by the pointer at 43-4 is then RUN. Consequently, if XYZ is loaded in at a location different from where the above line is stored, then the 64 will re-run the program line and search the tape for a second file 'XYZ'.

Files

Two types of file - program or data - may be written to or read from tape. However, the data transfer rate is too slow for data files to be used in many applications.

Bit	5	4	3	2	1	0
Configuration	1	0	1	1	1	1

Bit 3: Cassette Write
Bit 4: Cassette switch
Bit 5: Cassette motor

Figure 8.1 The 6510 Control Port
(location 0 : 1 = Output)

Three of the six lines from the 64 to the tape unit are routed through the 6510's control port. The sense line indicates whether the play, fast forward or rewind keys have been pressed. The motor line provides power to the cassette drive and the write line handles the data output stream. The other line of interest is the read line which is mapped to the C.I.A. 1 flag bit.

1	2	3	4	5	6
+	+	+	+	+	+
A	B	C	D	E	F
Ground	+5v	Motor	Read	Write	Sense

Figure 8.2 Cassette Port

The effective data transfer rate is about 300 baud (bits per second) although the actual transfer rate is more rapid. The difference arises primarily from each data byte being encoded twice to allow error checking procedures.

During a load sequence, a check is made on each byte using a parity bit. If a recoverable error is detected an adjustment is automatically made, with a more severe error causing an abort.

Although the data transfer rate is, by comparison with many home computers, very slow (several will reliably load and save files at up to 2400 baud with non-dedicated tape units) the Commodore configuration permits wide tolerance margins. As a result you can be fairly sure that software will load successfully, not just initially, but on the hundredth occasion.

Data transfer is via the tape buffer (locations 828-1019). As the normal activities of the VIC 2 cause many interruptions to the 6510s processing, the screen is blanked during input/output operations.

Program Protection

Often it may be necessary to locate a program file above the default position. For exampl,e to redefine part of the character set, it is a common practice to download a character set to between 2048 and 4095. The program start is moved prior to input, by placing a zero in the first location - in this case 4096 - before updating the BASIC pointer at 43-4. On power up, this contains &0801 (2049 decimal). To amend it to point to 4096, 16 is placed in location 44. The other system pointers are then reset with a NEW.

To SAVE and LOAD program files from/to a non standard location requires the use of a secondary address of 1 i.e. LOAD "XYZ",1,1 or SAVE "XYZ",1,1.

When loading a program in at a new location the adjustment sequence given above may omit the NEW command.

All very interesting, but what has it do do with program protection?

Unless an autorun sequence (see later) is used it is standard practice to lad and call a M/C program using a short BASIC program. An example of this form of the LOAD was given at the start of the chapter.

After the load has finished, the program pointed to by the vector in locations 43-4 is immediately RUN and any lines that follow the LOAD command are not executed.

Unless the program containing the LOAD command altered the vector to point to another BASIC program or the new program overlayed the original, the initial program is therefore re-run.

This loop is avoided by setting a flag on the first run (the variables are not reset):

```
10 IF X=1 THEN SYS 20000
20 X=1
30 LOAD "MACHINE CODE",1,1
```

This is the basis of a popular method of deterring the would be copier. A dummy file may be loaded or a file name ending with the clear screen character (together with a test for a blank screen) used. An alternative is to incorporate some form of secondary protection into the main program so that the program crashes or prints out a copyright message. On the 64, the ability to cover line listings using a series of deletes is extremely useful. Critical variable settings may be covered, as might sets of POKE commands. A useful variant of this is to place a POKE command in the program which places the value already held by a byte back in it. All is well until the listing is amended when the POKE causes a syntax error.

Going one step further involves the use of an autorun procedure. This should also negate the RUN/STOP/RESTORE options by changing the STOP routine vector at location 808 (&0328). An autostart after the initial program load is produced by changing the BASIC warm start vector to point to your machine code routine. To do this, the program must load into an area which covers the vector at 770-1. If your code starts in the free space from 681 then before saving the program change, the vector to hold &02A9 and include it in the SAVE.

On loading this back in, instead of the normal warm start routine being followed, your routine is executed.

If the BASIC program below is RUN after you have placed the code loader at &02A9 the vector is amended and saved with your routine. If you want to re-use the program then save it before use.

The RUN/STOP/RESTORE options are negated by placing 225 in location 808.

```
10 POKE 770,169:POKE 771,2 :REM ALTER WARM START TO POINT
CODE

20 POKE 43,167:POKE 44,2 :REM ALTER BASIC POINTER TO INCLUDE
THE CODE

30 POKE 45,4:POKE 46,3 :REM ALTER BASIC TOP POINTER

40 SAVE "AUTOSTART",1,1
```

This will work with either the LOAD or SHIFT/RUN/STOP sequences. To load and run a BASIC program starting at the default position in memory requires the code to place the commands in the keyboard queue before engaging the interpreter by a jump to &E394.

To examine the subject of program protection in more dtail would involve discussion of how to amend (via the cartridge auto-start) the sequence executed on a reset and the performance of logic operations on each byte or sections of the program together with decoding routines. This is beyond the scope of this book.

Although the subject of program protection attracts a great deal of interest the author feels it is somewhat overemphasised - if an expert wishes to examine a program there is no secure method of preventing him or her. The author well remembers watching a leading programmer remove, in half as many minutes, ten or so layers of protection he had put on a program.

More resident routines

At the end of chapter four, we referred to the routines available for operations on variables that use the floating point accumulators. Several of these make the lot of the machine language programmer a little more bearable.

For this section the two floating point accumulators are denoted by A1 and A2 and the 6510's accumulator by an A.

To place a two byte value in A1 in floating point form, the low order byte is placed in the A, and the high order byte in the Y register, before performing a JSR to GIVAYF at &B391. The value must be in the range 0-32767. A value outside this range will not affect the sign byte. For the reverse operation to convert a value in A1 to a positive two byute integer use the

AYINT routine at &B1BF. The two bytes are returned in locations 100 and 101 (low order byte).

For example, to place 1 in A1, load the Y register with 0 and the A with 1 before a call to GIVAYF. This yields:

Address	97	98	99	100	101	102
Value	137	128	0	0	0	0
	Exponent	Mantissa				

The exponent and mantissa may be copied from one accumulator to the other using either MOVAF (A1A2) at &BC0C or MOVFA (A2A1) at &BBFC.

These routines also copy the value held by the sign byte.

Transfers between the accumulators and memory are initialized by placing the low order byte of the address in the X register and the high order byte in the Y register. Further details of these routines together with those for addition, subtraction, multiplication and division are given in table 8.2.

Example: If &FB and &FC hold a pointer to the start of one row on the bit mapped screen, the following routine could be used to amend it to point to the first location of the row above by subtracting 320:

```
LDA &FB
LDY &FC
JSR 45969 Convert and place in A1
JSR 48140 Copy into A2
LDA =&40
LDY =1
JSR 45969 Place 320 in A1
JSR 47187 A1=A2-A1
JSR 45503 Convert to integer in 100 and 101
LDA 100
STA &FC
LDA 101
STA &FB Update pointer
```

One further routine which may be of interest to the reader is that used by the BASIC RND function. This routine at &E097, will place a pseudo-random number in A1 - if A1 is first initialized to zero.

Operation	Routine	Sources	Address
ADD	FADDT	A1 A2	B86A
ADD	FADD	A1 MEM	B867
SUBTRACT	FSUBT	A1-A2	B853
SUBTRACT	FSUB	MEM-A1	B950
MULTIPLY	FMULTT	A1*A2	BA2B
MULTIPLY	FMULT	A1*MEM	BA28
DIVIDE	FDIVT	A2/A1	BB12
DIVIDE	FDIVF	A2/MEM	BB07
DIVIDE	FDIV	MEM/A1	BB0F

Table 8.2 The principle floating point routines

APPENDIX A

DECIMAL - HEXADECIMAL TABLE

1..01	41..29	81..51	121..79	161..A1	201..C9	241..F1
2..02	42..2A	82..52	122..7A	162..A2	202..CA	242..F2
3..03	43..2B	83..53	123..7B	163..A3	203..CB	243..F3
4..04	44..2C	84..54	124..7C	164..A4	204..CC	244..F4
5..05	45..2D	85..55	125..7D	165..A5	205..CD	245..F5
6..06	46..2E	86..56	126..7E	166..A6	206..CE	246..F6
7..07	47..2F	87..57	127..7F	167..A7	207..CF	247..F7
8..08	48..30	88..58	128..80	168..A8	208..D0	248..F8
9..09	49..31	89..59	129..81	169..A9	209..D1	249..F9
10.0A	50..32	90..5A	130..82	170..AA	210..D2	250..FA
11.0B	51..33	91..5B	131..83	171..AB	211..D3	251..FB
12.0C	52..34	92..5C	132..84	172..AC	212..D4	252..FC
13.0D	53..35	93..5D	133..85	173..AD	213..D5	253..FD
14.0E	54..36	94..5E	134..86	174..AE	214..D6	254..FE
15.0F	55..37	95..5F	135..87	175..AF	215..D7	255..FF
16.10	56..38	96..60	136..88	176..B0	216..D8	
17.11	57..39	97..61	137..89	177..B1	217..D9	
18.12	58..3A	98..62	138..8A	178..B2	218..DA	
19.13	59..3B	99..63	139..8B	179..B3	219..DB	
20.14	60..3C	100..64	140..8C	180..B4	220..DC	
21.15	61..3D	101..65	141..8D	181..B5	221..DD	
22.16	62..3E	102..66	142..8E	182..B6	222..DE	
23.17	63..3F	103..67	143..8F	183..B7	223..DF	
24.18	64..40	104..68	144..90	184..B8	224..E0	
25.19	65..41	105..69	145..91	185..B9	225..E1	
26.1A	66..42	106..6A	146..92	186..BA	226..E2	
27.1B	67..43	107..6B	147..93	187..BB	227..E3	
28.1C	68..44	108..6C	148..94	188..BC	228..E4	
29.1D	69..45	109..6D	149..95	189..BD	229..E5	
30.1E	70..46	110..6E	150..96	190..BE	230..E6	
31.1F	71..47	111..6F	151..97	191..BF	231..E7	
32.20	72..48	112..70	152..98	192..C0	232..E8	
33.21	73..49	113..71	153..99	193..C1	233..E9	
34.22	74..4A	114..72	154..9A	194..C2	234..EA	
35.23	75..4B	115..73	155..9B	195..C3	235..EB	
36.24	76..4C	116..74	156..9C	196..C4	236..EC	
37.25	77..4D	117..75	157..9D	197..C5	237..ED	
38.26	78..4E	118..76	158..9E	198..C6	238..EE	
39.27	79..4F	119..77	159..9F	199..C7	239..EF	
40.28	80..50	120..78	160..A0	200..C8	240..F0	

APPENDIX B

COLOUR TABLE

0	BLACK
1	WHITE
2	RED
3	CYAN
4	PURPLE
5	GREEN
6	BLUE
7	YELLOW
8	ORANGE
9	BROWN
10	L. RED
11	GREY 1
12	GREY 2
13	L. GREEN
14	L. BLUE
15	GREY 3

APPENDIX C

6510 INSTRUCTIONS

ADC	Add with carry	JSR	Jump to subroutine
AND	Logical AND	LDA	Load accumulator
ASL	Arithmetic shift left	LDX	Load the X register
BCC	Branch on carry clear	LDY	Load the Y register
BCS	Branch on carry set	LSR	Logical shift right
BEQ	Branch on zero data	NOP	No operation
BIT	Test if equal	ORA	OR. with data
BMI	Branch on minus	PHA	Push A. on stack
BNE	Branch if not zero	PHP	Push P.S.R. on stack
BPL	Branch on plus	PLA	Pull stack to A.
BRK	Break	PLP	Pull stack to P.S.R.
BVC	Branch on V clear	ROL	Rotate 1 position left
BVS	Branch on V set	ROR	Rotate 1 position right
CLC	Clear carry flag	RTI	Return from interrupt
CLD	Clear decimal flag	RTS	Return from subroutine
CLI	Clear interrupt flag	SBC	Subtract with carry
CLV	Clear overflow flag	SEC	Set carry flag
CMP	Compare data to A.	SED	Set decimal flag
CPX	Compare data to X	SEI	Set interrupt flag
CPY	Compare data to Y	STA	Store accumulator
DEC	Decrement memory 1	STX	Store X register
DEX	Decrement X register	STY	Store Y register
DEY	Decrement Y register	TAX	Transfer A. to X
EOR	Exclusive OR data & A.	TAY	Transfer A. to Y
INC	Increment memory 1	TSX	Transfer the S.P. to X
INX	Increment X register	TXA	Transfer X to A.
INY	Increment Y register	TXS	Transfer X to the S.P.
JMP	Jump	TYA	Transfer Y to A.

APPENDIX D

KERNAL ROUTINES

INPUT / OUTPUT

INITIALISE :	SETLFS	&FFBA	LOAD	:	&FFD5
	SETNAM	&FFBD			
	OPEN	&FFC0	SAVE	:	&FFD8
SERIAL :	TALK	&FFB4	TKSA		&FF96
	LISTEN	&FFB1	SECOND		&FF93
	UNTLK	&FFAB	UNLSN		&FFAE
	ACPTR	&FFA5	CIOUT		&FFA8
	SETTMO	&FFA2			
GENERAL :	IOINIT	&FF84	CLRCHN		&FFCC
	READST	&FFB7	IOBASE		&FFF3
	RESTOR	&FF8A	VECTOR		&FF8D
	CHKIN	&FFC6	CHKOUT		&FFC9
	CHRIN	&FFCF	CHROUT		&FFD2
	CLALL	&FFE7	CLOSE		&FFC3
KEYBOARD :	GETIN	&FFE4	SCNKEY		&FF9F
	STOP	&FFE1	UDTIM		&FFEA

SYSTEM

CINT	&FF81	MEMBOT	&FF9C
MEMTOP	&FF99	PLOT	&FFF0
RAMTAS	&FF87	RDTIM	&FFDE
SCREEN	&FFED	SETMSG	&FF90
SETTIM	&FFD8		

APPENDIX E

MEMORY MAP

DECIMAL	HEX	DESCRIPTION
0	0000	6510 I/O Port Direction register
1	0001	6510 I/O Port
2	0002	Unused
3-4	0003-0004	Vector: F.P. to Int. conversion
5-6	0005-0006	Vector: Int. to F.P. conversion
7	0007	Search character
8	0008	Flag: Scan, end of string
9	0009	TAB column
10	000A	Flag: 1 Verify, 0 Load
11	000B	Input buffer pointer
12	000C	Flag: Default DIM value
13	000D	Data T. string &FF numeric &00
14	000E	Data T. Int. &80 F.P. &00
15	000F	Flag: Garbage C./data scan/LIST
16	0010	Flag: FN, Subscript
17	0011	Flag: INPUT, GET, READ
18	0012	Flag: TAN sign, Comparison
19	0013	Flag: Input cue
20-1	0014-0015	Temporary integer storage
22	0016	Temporary string stack pointer
23-4	0017-0018	Pointer to last temporary string
25-33	0019-0021	Stack for temp. strings
34-7	0022-0025	Utility pointer area
38-42	0026-002A	F.P. result - Multiplication
43-4	002B-002C	Pointer to start of BASIC program
45-6	002D-002E	Pointer to BASIC variables
47-8	002F-0030	Pointer to BASIC arrays
49-50	0031-0032	Pointer to end of arrays + 1
51-2	0033-0034	Pointer to string storage
53-4	0035-0036	Utility string pointer
55-6	0037-0038	Pointer to BASIC ceiling
57-8	0039-003A	Current BASIC line number
59-60	003B-003C	Previous BASIC line number

Decimal	*Hex*	*Description*
61-2	003D-003E	Pointer BASIC CONT
63-4	003F-0040	Current DATA line number
65-6	0041-0042	Location of present DATA item
67-8	0043-0044	INPUT routine vector
69-70	0045-0046	Current BASIC variable
71-2	0047-0048	Current variable address
73-4	0049-004A	Pointer to index of FOR/NEXT
75-96	004B-0060	Scratch-pad
97-101	0061-0065	F.P. Accumulator £ 1
102	0066	Sign F.P.A. £ 1
103	0067	Pointer to series eval. constant
104	0068	F.P.A. £ 1 M.S.B. Overflow
105-9	0069-006D	F.P. Accumulator £ 2
110	006E	Sign F.P.A. £ 2
111	006F	Sign comparison F.P.A. £1 - £2
112	0070	F.P.A. £ 1 Low order,rounding
113-4	0071-0072	Cassette buffer pointer
115-138	0073-008A	CHRGET routine
139-143	008B-008F	F.P.seed value for RND
144	0090	I/O Status ST
145	0091	Flag : STOP/RVS keys
146	0092	Constant for tape timing
147	0093	Flag : 0 Load, 1 Verify
148	0094	Flag : Serial output byte buff'd
149	0095	Serial output byte
150	0096	Tape op. st. value
151	0097	Scratch-pad
152	0098	Current No. of open files
153	0099	Input default device number
154	009A	Output default device number
155	009B	Tape character parity
156	009C	Flag : Tape byte input
157	009D	Flag : &80 Direct, &00 Program
158	009E	Tape error - Pass 1
159	009F	Pass 2
160-2	00A0-00A2	System clock
163	00A3	Serial bit count
164	00A4	Cycle count
165	00A5	Tape output bit count
166	00A6	Pointer to tape I/O buffer
167-8	00A7-00A8	Scratch-pad
169	00A9	Flag : RS-232 Start bit chk.
170	00AA	RS-232 input byte/Tape s.p.

Decimal	*Hex*	*Description*
171	00AB	RS-232 parity/Tape s.p.
172-3	00AC-00AD	Pointer : Tape buffer/Scrolling
174-5	00AE-00AF	Tape end addresses, marker
176-7	00B0-00B1	More tape timing constants
178-9	00B2-00B3	Pointer to Tape buffer
180-1	00B4-00B5	RS-232 / Tape s.p.
182	00B6	RS-232 output buffer
183	00B7	File name length
184	00B8	Logical file number
185	00B9	Secondary address
186	00BA	Device number
187-8	00BB-00BC	Pointer to file name
189	00BD	RS-232/Tape s.p.
190	00BE	Tape blocks outstanding to R/W
191	00BF	Serial word buffer
192	00C0	Tape motor interlock
193-4	00C1-00C2	I/O start location
195-6	00C3-00C4	Tape s.p.
197	00C5	Last key pressed
198	00C6	Keyboard queue character count
199	00C7	Flag : RVS characters
200	00C8	Pointer to end of log.line..INPUT
201-2	00C9-00CA	Row & column of cursor @ INPUT
203	00CB	Flag : Shift key
204	00CC	Enable : Cursor flash
205	00CD	Cursor toggle countdown
206	00CE	Character under the cursor
207	00CF	Flag : Cursor blink status
208	00D0	Flag : GET / INPUT
209-210	00D1-00D2	Pointer to current screen line
211	00D3	Cursor column
212	00D4	Flag : Editor mode, &00 = default
213	00D5	Screen line length
214	00D6	Cursor physical line number
215	00D7	Scratch-pad
216	00D8	Inserts outstanding
217-242	00D9-00F2	Screen line link table/Editor s.p.
243-4	00F3-00F4	Pointer to screen colour RAM
245-6	00F5-00F6	Start address of keyboard table
247-8	00F7-00F8	Pointer to RS-232 input buffer
249-250	00F9-00FA	" " output "
251-4	00FB-00FE	NOT USED
255	00FF	BASIC s.p.

Decimal	*Hex*	*Description*
256-511	0100-01FF	6510 STACK
512-600	0200-0258	BASIC buffer
601-610	0259-0262	Open file table
611-620	0263-026C	File device table
621-630	026D-0276	Secondary address table
631-640	0277-0280	Keyboard queue
641-2	0281-0282	BASIC start
643-4	0283-0284	Top of BASIC memory
645	0285	Flag : Timeout on serial bus
646	0286	Foreground colour
647	0287	Cursor background colour
648	0288	Top of screen memory
649	0289	Keyboard buffer capacity
650	028A	Flag : Key auto-repeat
651	028B	Repeat speed counter
652	028C	Repeat delay counter
653	028D	Flag : Shift / CTRL / C= Keys
654	028E	Keyboard shift pattern
655-6	028F-0290	Pointer to keyboard table setup
657	0291	Flag : Shift keys enabled
658	0292	Flag : Scroll, 0 = enabled
659	0293	Duplicate 6551 ctrl. reg. RS-232
660	0294	" " comm. "
661-2	0295-0296	Timing RS-232
663	0297	Duplicate 6551 status reg.RS-232
664	0298	RS-232 : No of bits O/S to output
665-6	0299-029A	RS-232 Baud
667	029B	RS-232 Index to End of I. buffer
668	029C	RS-232 Input buffer start
669	029D	RS-232 Output buffer start
670	029E	RS -232 Index to End of O. buffer
671-2	029F-02A0	IRQ vector during tape I/O
673	02A1	RS-232 NMI Enables
674	02A2	TOD sense during tape I/O
675	02A3	Tape I/O s.p.
676	02A4	Flag : Timer A, C.I.A.1 Enabled
677	02A5	Line index s.p.
678	02A6	Flag : PAL / NTSC
679-767	02A7-02FF	UNUSED
768-9	0300-0301	Vector : BASIC error message
770-1	0302-0303	Vector : BASIC warm start
772-3	0304-0305	Vector : Tokenize BASIC

Decimal	*Hex*	*Description*
774-5	0306-0307	Vector : LIST
776-7	0308-0309	Vector : BASIC code dispatch
778-9	030A-030B	Vector : BASIC token eval.
780	030C	Temporary accumulator storage
781	030D	Temporary X register storage
782	030E	Temporary Y register storage
783	030F	Temporary P.S.R. storage
784	0310	&4C : Jump opcode for USR
785-6	0311-0312	Address of USR
787	0313	UNUSED
788-9	0314-0315	Vector : Hardware IRQ
790-1	0316-0317	Vector : BRK
792-3	0318-0319	Vector : NMI
794-5	031A-031B	Vector : OPEN
796-7	031C-031D	Vector : CLOSE
798-9	031E-031F	Vector : CHKIN
800-1	0320-0321	Vector : CHKOUT
802-3	0322-0323	Vector : CLRCHN
804-5	0324-0325	Vector : CHRIN
806-7	0326-0327	Vector : CHROUT
808-9	0328-0329	Vector : STOP
810-1	032A-032B	Vector : GETIN
812-3	032C-032D	Vector : CLALL
814-5	032E-032F	Vector : USR
816-7	0330-0331	Vector : LOAD
818-9	0332-0333	Vector : SAVE
820-827	0334-033B	UNUSED
828-1019	033C-03FB	Tape buffer
1020-1023	03FC-03FF	UNUSED

APPENDIX F

SPRITE TABLE

Pointers	2040-7	07F8-07FF
Locations	53248-53263	D000-D00F
M..B. of X	53264	D010
Expand Y	53271	D017
Collision	53273	D019
Priority to Background	53275	D01B
Multi-colour	53276	D01C
Expand X	53277	D01D
S - S Coll'n	53278	D01E
S - B Coll'n	53279	D01F
Multi-Colour	53286	D026
Colour 0	53287	D027
1	53288	D028
2	53289	D029
3	53290	D02A
4	53291	D02B
5	53292	D02C
6	53293	D02D
7	53294	D02E

APPENDIX G

V.I.C. REGISTERS

53248	-	53294
D000	-	D02E

53265 **Control Register 1**

MSB Raster R/W /	*Ext'd Colour T.*	*/ Bit Map*	*/ Blank Screen*
8	7	6	5
24/5 Row Text	*Smooth Scroll Position Y*		
4	3	2	1

53266 R / W Raster Value

53267-8 Light pen latches

53270 Control Register 2

Unused	*/ Unused*	*/ CLEAR*	*/ Multi-colour mode*
8	7	6	5

38/40 Column /	*Smooth Scroll Position X*		
4	3	2	1

53272 Memory Control Register

Bits 8 - 4 Scan base address 3 - 2 Dot base address

53273-4 Interrupt Registers - see chapter 6

53280 Border colours

APPENDIX H

C.I.A. # 1

56320	-	56575
DC00	-	DCFF

56320 Data Port A

56321 Data Port B

56322 Direction Reg. Port A

56323 Direction Reg. B

56324-5 Timer A

56326-7 Timer B

56328-56331 Real Time Clock

56332 Serial I / O Buffer

56333 Interrupt Control Register (IRQ) - see chapter 6

56334 Control Register A

R.T.C. 50/60 Hz	*/ In/Output S.Port*	*/ Timer A units*
8	7	6
Force load T.A	*/ T.A Run mode*	*/ T.A Output mode*
5	4	3

T.A Output to P.B6 / Timer A Start/Stop

2	1

56335 Control Register B

R.T.C. Alarm/Clock set / Timer B Unit mode select

8	7/6

As in Control register A, but read timer B for A

---

5 - 1

C.I.A. # 2

56576	-	56831
DD00	-	DDFF

56576 Data Port A

56577 Data Port B

56578 Data Direction A

56579 Data Direction B

56580/1 Timer A

56582/3 Timer B

56584-7 Real Time Clock

56588 Serial I/O buffer

56589 Interrupt Control Register (NMI) - see chapter 6

56590 Control Register A - As for C.I.A. # 1

56591 Control Register B - As for C.I.A. # 1

Notes

1 The IRQ LINE FROM C.I.A. # 2 initiates NMI's

2 The M.S. bit of the R.T. clock is the AM / PM flag

3 Timer A unit mode is either phase 2 system pulses or positive CNT transitions. Timer B may also count Timer A underflow pulses.

4 The two low order bits of location 56576 select the 16K video bank if the corresponding bits in the data direction register (56578) are set to output.

APPENDIX I

I / O Error Messages / Device Status

SERIAL	:	0	......	Time out - Write
		1	......	Time out - Read
		64	......	EOI line
		128	......	No device

Tape	:	**Read**		
		4	......	Short block
		8	......	Long block
		16	......	Fatal Read error
		32	......	Checksum error
		64	......	File end
		128	......	Tape end

Tape : Verify / Load

4	......	Short block

8	......	Long block
16/32	......	Checksum / Mismatch Error
128	......	Tape end

Note : Values returned by the READST routine in the A.

INDEX

A

absolute address, 53,54,55
accumulator, 12
ADC, 19,25
address bus, 9
address enable line, 12
address instruction, 17,18,53
AND, 7,25,76
arithmetic operations, 19
animation, 76
ASCII, 111
ASL, 26
assembler, 15,16,78

B

BASIC interpreter, 3,109
BASIC program structure, 109
BCD, 58
binary notation, 4
binary fractions, 59
binary search, 67
BIT, 20
bit map, 91
block manipulation, 66
branch instructions, 21
BRK, 63

C

carry flag, 21,60
cassette handling 117
character ROM, 74
CHRGET, 75
CHRIN, 74
C.I.A., 9,14,69,
CINT, 115
CIOUT, 113
CLALL, 114
CLC, 17
clock - system, 10,114
CLOSE, 114
CLRCHIN, 114
compare instruction, 21
control & C= keys, 74

D

data bus, 9
data transmission, 15
decrement instruction, 19
Direct Memory Access, 10
direct mode, 2

E

elements - data, 66
EOR, 7
exponent, 59

F

file, 111,118
flag bits, 12
FLAG bit, 73
floating point, 59,111
function keys, 75

G

general purpose registers, 2,3
GETIN, 74
GIVAYF, 121
graphics, 80

H

handshaking, 15,113
hexadecimal, 6
Hex Loader, 78
high level languages, 3
high order byte, 53

I

immediate addressing, 54
implied addressing, 54
increment instruction, 19
indirect addressing, 55,56
input / output, 9,111
input / output port - 6510, 15
instructions-processor, 2,17,25
interrupts, 10,63,72
interrupt flag, 63
IRQ, 63

J

JMP, 23,56
joysticks, 112
JSR, 23

K

KERNAL, 109
keyboard, 69
keyboard queue, 70
keywords, 109

L

LIST protection, 109
LISTEN, 113
LOAD, 112
logical operations, 7,20
low order byte, 54
Low resolution, 86
L.S.B., 5
LSR, 20

M

Mantissa, 59,110
mask registers, 72
MEMBOT, 115
memory, 1
memory operations, 18
memory mapping, 9
MEMTOP, 115
microprocessors, 1
MOVAF, MOVFA, 122
M.S.B., 5
multicolour, 93
multiple precision, 60
multiplication, 20

N

nibble, 6
NOP, 66
NMI, 63

O

opcode, 17
OPEN, 111
operating system, 3
OR, 7
overflow flag, 60

P

page, 2
parallel I/O, 15
P.C., 2,12,22,63
PLOT, 115
polling, 73
pop, 64
ports, 113
positional independence, 67
program counter - see PC
program structure, 66
P.S.R., 2,21,
push, 64

R

R.A.M., 2
RAMTAS, 115
raster register, 73
RDTIM, 114
READST, 116
RESTOR, 116
R/W line, 12
RESET, 64
ROL, 20
R.O.M., 2
ROR, 20
RS-232, 112
RTI, 63
RTS, 23

S

SAVE, 112
SBC, 20
SCNKEY, 73,74
SED, 59
SCREEN, 115
scrolling, 93
serial I/O, 14
serial port, 113
serial search, 67
SETLFS, 111
SETMSG, 115
SETNAM, 111
SETTIM, 114
shift key, 74
S.I.D., 9
single pass assembler, 18
split screen, 94
sprites, 76
sprite collisions, 73
stack, 12,21,64
stack instructions, 64
stack pointer, 13
STOP key, 74
strings, 110
subroutines, 23,64
subtraction, 5,60

T

TALK, 113
Timers C.I.A., 73
timing reference, 9
TKSA, 113
tokens, 109
TSX, 65
Two's complement, 5,60
TXS, 65

U

UDTIM, 74
UNLSN, 113
UNTALK, 114
USR function, 61
User port, 113

V

Variables:
Integer, 59,110
floating point, 59,60,110
VECTOR, 116
vectors-interrupt, 63
V.I.C. 2,9,10,72,76

W

wedge, 75

Z

Z flag, 21
Zero page addressing, 54
Zero page locations, 66

SPEEDING UP YOUR 64!

When you first got your hands on a computer, it seemed amazingly fast. So fast that you didn't have time to think as it whizzed through your first BASIC program --- leastways, the first one you wrote that worked!

But when you began writing bigger programs, especially those that were of the fast-action arcade variety, or any that were supposed to respond immediately to your merest whim, the weaknesses of your '64' was exposed. Yes, it was too slow for that final big project.

Before you consign it to the broom cupboard, just see how machine code would change your life. Liberated from BASIC, your programs really will be as fast as you want them to be. Machine code is limited only by the speed of the microprocessor in your '64' and, since that executes hundreds of thousands of instructions a second, it's sure to be fast enough.

The book explaines machine code in an easy manner, for anyone who can program their '64' in BASIC. The contents include:

- *Microprocessors. High-level languages. Binary & Hexadecimal representation. Logical operations.
- *The 6510 Microprocessor. System organisation. The 6510 Architecture.
- *Memory usage. Input/Output operations.
- *Instruction framework. Comparison and Branch Instructions. Instruction menu.
- *Addressing Techniques
- *The indirect approach.
- *Stack & Subroutine use.
- *Screen & Keyboard Techniques; The C. I. A. 1. The interrupt routines. Function keys. Sprites & Animation.
- *The Commodore 64 R. O. M.;. Input/Output. The Kernal.

An important feature of this book is that it explains how machine code programs still use such important features of the '64' as graphics and sprites.

This is just one of the excellent Sigma books available on all aspects of modern micros. Write today for a catalogue, or tell us about a book that you would like to write.

Sigma Press
5 Alton Road
Wilmslow
Cheshire
SK9 5DY

Price £6.95

ISBN 0 905104 91 9